XML Schema Part 0: Primer
# Table of Contents

**XML Schema Part 0: Primer**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3C Recommendation, 2 May 2001</td>
<td>1</td>
</tr>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>Status of this document</td>
<td>1</td>
</tr>
<tr>
<td>Table of contents</td>
<td>2</td>
</tr>
<tr>
<td>Appendices</td>
<td>3</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2 Basic Concepts: The Purchase Order</td>
<td>4</td>
</tr>
<tr>
<td>2.1 The Purchase Order Schema</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Complex Type Definitions, Element &amp; Attribute Declarations</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Simple Types</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Anonymous Type Definitions</td>
<td>15</td>
</tr>
<tr>
<td>2.5 Element Content</td>
<td>16</td>
</tr>
<tr>
<td>2.6 Annotations</td>
<td>19</td>
</tr>
<tr>
<td>2.7 Building Content Models</td>
<td>19</td>
</tr>
<tr>
<td>2.8 Attribute Groups</td>
<td>21</td>
</tr>
<tr>
<td>2.9 Nil Values</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Target Namespaces &amp; Unqualified Locals</td>
<td>24</td>
</tr>
<tr>
<td>3.2 Qualified Locals</td>
<td>26</td>
</tr>
<tr>
<td>3.3 Global vs. Local Declarations</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Undeclared Target Namespaces</td>
<td>29</td>
</tr>
<tr>
<td>4. Advanced Concepts II: The International Purchase Order</td>
<td>30</td>
</tr>
<tr>
<td>4.1 A Schema in Multiple Documents</td>
<td>30</td>
</tr>
<tr>
<td>4.2 Deriving Types by Extension</td>
<td>33</td>
</tr>
<tr>
<td>4.3 Using Derived Types in Instance Documents</td>
<td>33</td>
</tr>
<tr>
<td>4.4 Deriving Complex Types by Restriction</td>
<td>34</td>
</tr>
<tr>
<td>4.5 Redefining Types &amp; Groups</td>
<td>36</td>
</tr>
<tr>
<td>4.6 Substitution Groups</td>
<td>37</td>
</tr>
<tr>
<td>4.7 Abstract Elements and Types</td>
<td>38</td>
</tr>
<tr>
<td>4.8 Controlling the Creation &amp; Use of Derived Types</td>
<td>39</td>
</tr>
<tr>
<td>5.1 Specifying Uniqueness</td>
<td>43</td>
</tr>
<tr>
<td>5.2 Defining Keys &amp; their References</td>
<td>44</td>
</tr>
<tr>
<td>5.3 XML Schema Constraints vs. XML 1.0 ID Attributes</td>
<td>44</td>
</tr>
<tr>
<td>5.4 Importing Types</td>
<td>44</td>
</tr>
<tr>
<td>5.5 Any Element, Any Attribute</td>
<td>47</td>
</tr>
<tr>
<td>5.6 schemaLocation</td>
<td>50</td>
</tr>
<tr>
<td>5.7 Conformance</td>
<td>51</td>
</tr>
<tr>
<td>A Acknowledgements</td>
<td>52</td>
</tr>
<tr>
<td>B Simple Types &amp; their Facets</td>
<td>52</td>
</tr>
<tr>
<td>C Using Entities</td>
<td>55</td>
</tr>
<tr>
<td>D Regular Expressions</td>
<td>56</td>
</tr>
<tr>
<td>E Index</td>
<td>57</td>
</tr>
</tbody>
</table>

**XML Schema Part 1: Structures**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3C Recommendation 2 May 2001</td>
<td>60</td>
</tr>
<tr>
<td>Abstract</td>
<td>60</td>
</tr>
</tbody>
</table>
## Table of Contents

Status of this document..................................................................................................................60
Table of contents..........................................................................................................................61
Appendices....................................................................................................................................61

### 1 Introduction..........................................................................................................................61
   1.1 Purpose..................................................................................................................................62
   1.2 Dependencies on Other Specifications..............................................................................62
   1.3 Documentation Conventions and Terminology...............................................................62

### 2 Conceptual Framework.........................................................................................................64
   2.1 Overview of XML Schema.................................................................................................64
   2.2 XML Schema Abstract Data Model....................................................................................65
   2.3 Constraints and Validation Rules.....................................................................................71
   2.4 Conformance.....................................................................................................................71
   2.5 Names and Symbol Spaces..............................................................................................72
   2.6 Schema–Related Markup in Documents Being Validated.............................................73
   2.7 Representation of Schemas on the World Wide Web.......................................................73

### 3 Schema Component Details................................................................................................74
   3.1 Introduction.......................................................................................................................74
   3.2 Attribute Declarations.......................................................................................................77
   3.3 Element Declarations.......................................................................................................85
   3.4 Complex Type Definitions...............................................................................................98
   3.5 AttributeUses..................................................................................................................113
   3.6 Attribute Group Definitions............................................................................................115
   3.7 Model Group Definitions.................................................................................................117
   3.8 Model Groups..................................................................................................................120
   3.9 Particles............................................................................................................................124
   3.10 Wildcards........................................................................................................................130
   3.11 Identity–constraint Definitions.......................................................................................134
   3.12 Notation Declarations....................................................................................................141
   3.13 Annotations.....................................................................................................................144
   3.14 Simple Type Definitions...............................................................................................146
   3.15 Schemas as a Whole.......................................................................................................152

### 4 Schemas and Namespaces: Access and Composition..........................................................159
   4.1 Layer 1: Summary of the Schema–validity Assessment Core........................................160
   4.2 Layer 2: Schema Documents, Namespaces and Composition........................................161
   4.3 Layer 3: Schema Document Access and Web–interoperability......................................168

### 5 Schemas and Schema–validity Assessment........................................................................170
   5.1 Errors in Schema Construction and Structure...............................................................170
   5.2 Assessing Schema–Validity............................................................................................171
   5.3 Missing Sub–components.................................................................................................172
   5.4 Responsibilities of Schema–aware Processors...............................................................172

### A Schema for Schemas (normative).....................................................................................172

### B References (normative)......................................................................................................192

### C Outcome Tabulations (normative)....................................................................................193
   C.1 Validation Rules..............................................................................................................193
   C.2 Contributions to the post–schema–validation infoset......................................................194
   C.3 Schema Representation Constraints..............................................................................195
   C.4 Schema Component Constraints...................................................................................196

### D Required Information Set Items and Properties (normative)..........................................199
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E Schema Components Diagram (non−normative).</td>
<td>200</td>
</tr>
<tr>
<td>F Glossary (non−normative).</td>
<td>202</td>
</tr>
<tr>
<td>G DTD for Schemas (non−normative).</td>
<td>205</td>
</tr>
<tr>
<td>H Analysis of the Unique Particle Attribution Constraint (non−normative)</td>
<td>211</td>
</tr>
<tr>
<td>I References (non−normative).</td>
<td>212</td>
</tr>
<tr>
<td>J Acknowledgements (non−normative).</td>
<td>213</td>
</tr>
<tr>
<td><strong>XML Schema Part 2: Datatypes</strong></td>
<td>215</td>
</tr>
<tr>
<td>W3C Recommendation 02 May 2001.</td>
<td>215</td>
</tr>
<tr>
<td>Abstract.</td>
<td>215</td>
</tr>
<tr>
<td>Status of this document.</td>
<td>215</td>
</tr>
<tr>
<td>Table of contents.</td>
<td>216</td>
</tr>
<tr>
<td>Appendices.</td>
<td>216</td>
</tr>
<tr>
<td>1 Introduction.</td>
<td>216</td>
</tr>
<tr>
<td>1.1 Purpose.</td>
<td>216</td>
</tr>
<tr>
<td>1.2 Requirements.</td>
<td>217</td>
</tr>
<tr>
<td>1.3 Scope.</td>
<td>217</td>
</tr>
<tr>
<td>1.4 Terminology.</td>
<td>217</td>
</tr>
<tr>
<td>1.5 Constraints and Contributions.</td>
<td>218</td>
</tr>
<tr>
<td>2 Type System.</td>
<td>218</td>
</tr>
<tr>
<td>2.1 Datatype.</td>
<td>218</td>
</tr>
<tr>
<td>2.2 Value space.</td>
<td>218</td>
</tr>
<tr>
<td>2.3 Lexical space.</td>
<td>219</td>
</tr>
<tr>
<td>2.4 Facets.</td>
<td>220</td>
</tr>
<tr>
<td>2.5 Datatype dichotomies.</td>
<td>220</td>
</tr>
<tr>
<td>3 Built−in datatypes.</td>
<td>224</td>
</tr>
<tr>
<td>3.1 Namespace considerations.</td>
<td>226</td>
</tr>
<tr>
<td>3.2 Primitive datatypes.</td>
<td>226</td>
</tr>
<tr>
<td>3.3 Derived datatypes.</td>
<td>243</td>
</tr>
<tr>
<td>4 Datatype components.</td>
<td>258</td>
</tr>
<tr>
<td>4.1 Simple Type Definition.</td>
<td>258</td>
</tr>
<tr>
<td>4.2 Fundamental Facets.</td>
<td>265</td>
</tr>
<tr>
<td>4.3 Constraining Facets.</td>
<td>269</td>
</tr>
<tr>
<td>5 Conformance.</td>
<td>288</td>
</tr>
<tr>
<td>A Schema for Datatype Definitions (normative).</td>
<td>289</td>
</tr>
<tr>
<td>B DTD for Datatype Definitions (non−normative).</td>
<td>309</td>
</tr>
<tr>
<td>C Datatypes and Facets.</td>
<td>313</td>
</tr>
<tr>
<td>C.1 Fundamental Facets.</td>
<td>313</td>
</tr>
<tr>
<td>D ISO 8601 Date and Time Formats.</td>
<td>314</td>
</tr>
<tr>
<td>D.1 ISO 8601 Conventions.</td>
<td>314</td>
</tr>
<tr>
<td>D.2 Truncated and Reduced Formats.</td>
<td>315</td>
</tr>
<tr>
<td>D.3 Deviations from ISO 8601 Formats.</td>
<td>316</td>
</tr>
<tr>
<td>E Adding durations to dateTimes.</td>
<td>316</td>
</tr>
<tr>
<td>E.1 Algorithm.</td>
<td>317</td>
</tr>
<tr>
<td>E.2 Commutativity and Associativity.</td>
<td>319</td>
</tr>
<tr>
<td>F Regular Expressions.</td>
<td>319</td>
</tr>
<tr>
<td>F.1 Character Classes.</td>
<td>320</td>
</tr>
<tr>
<td>G Glossary (non−normative).</td>
<td>329</td>
</tr>
</tbody>
</table>
# Table of Contents

H References.......................................................................................................................................332  
  H.1 Normative.................................................................................................................................332  
  H.2 Non–normative.........................................................................................................................332  
I Acknowledgements (non–normative)...............................................................................................334  

**XML Schema Requirements**........................................................................................................336  
  W3C Note 15 February 1999........................................................................................................336  
  Table of Contents..........................................................................................................................336  
  1. Overview...................................................................................................................................337  
  2. Purpose.....................................................................................................................................337  
  3. Usage Scenarios.........................................................................................................................338  
  4. Design Principles......................................................................................................................339  
  5. Requirements............................................................................................................................339  
    Structural requirements.............................................................................................................339  
    Datatype requirements.............................................................................................................340  
    Conformance..............................................................................................................................340
XML Schema Part 0: Primer

W3C Recommendation, 2 May 2001

This version:
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

Latest version:
http://www.w3.org/TR/xmlschema-0/

Previous version:
http://www.w3.org/TR/2001/PR-xmlschema-0-20010330/

Editor:
David C. Fallside (IBM) fallside@us.ibm.com

Abstract

XML Schema Part 0: Primer is a non-normative document intended to provide an easily readable description of the XML Schema facilities, and is oriented towards quickly understanding how to create schemas using the XML Schema language. XML Schema Part 1: Structures and XML Schema Part 2: Datatypes provide the complete normative description of the XML Schema language. This primer describes the language features through numerous examples which are complemented by extensive references to the normative texts.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document has been produced by the W3C XML Schema Working Group as part of the W3C XML Activity. The goals of the XML Schema language are discussed in the XML Schema Requirements document. The authors of this document are the members of the XML Schema Working Group. Different parts of the document have different editors.

This version of this document incorporates some editorial changes from earlier versions.

Please report errors in this document to www-xml-schema-comments@w3.org (archive). The list of known errors in this specification is available at http://www.w3.org/2001/05/xmlschema-errata.

The English version of this specification is the only normative version. Information about translations of this document is available at http://www.w3.org/2001/05/xmlschema-translations.
A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

Table of contents

1 Introduction
2 Basic Concepts: The Purchase Order
   2.1 The Purchase Order Schema
   2.2 Complex Type Definitions, Element & Attribute Declarations
      2.2.1 Occurrence Constraints
      2.2.2 Global Elements & Attributes
      2.2.3 Naming Conflicts
   2.3 Simple Types
      2.3.1 List Types
      2.3.2 Union Types
   2.4 Anonymous Type Definitions
   2.5 Element Content
      2.5.1 Complex Types from Simple Types
      2.5.2 Mixed Content
      2.5.3 Empty Content
      2.5.4 anyType
   2.6 Annotations
   2.7 Building Content Models
   2.8 Attribute Groups
   2.9 Nil Values
3 Advanced Concepts I: Namespaces, Schemas & Qualification
   3.1 Target Namespaces & Unqualified Locals
   3.2 Qualified Locals
   3.3 Global vs. Local Declarations
   3.4 Undeclared Target Namespaces
4 Advanced Concepts II: The International Purchase Order
   4.1 A Schema in Multiple Documents
   4.2 Deriving Types by Extension
   4.3 Using Derived Types in Instance Documents
   4.4 Deriving Complex Types by Restriction
   4.5 Redefining Types & Groups
   4.6 Substitution Groups
   4.7 Abstract Elements & Types
   4.8 Controlling the Creation & Use of Derived Types
5 Advanced Concepts III: The Quarterly Report
   5.1 Specifying Uniqueness
   5.2 Defining Keys & their References
   5.3 XML Schema Constraints vs. XML 1.0 ID Attributes
   5.4 Importing Types
      5.4.1 Type Libraries
   5.5 Any Element, Any Attribute
   5.6 schemaLocation
   5.7 Conformance
1 Introduction

This document, XML Schema Part 0: Primer, provides an easily approachable description of the XML Schema definition language, and should be used alongside the formal descriptions of the language contained in Parts 1 and 2 of the XML Schema specification. The intended audience of this document includes application developers whose programs read and write schema documents, and schema authors who need to know about the features of the language, especially features that provide functionality above and beyond what is provided by DTDs. The text assumes that you have a basic understanding of XML 1.0 and XML-Namespaces. Each major section of the primer introduces new features of the language, and describes those features in the context of concrete examples.

Section 2 covers the basic mechanisms of XML Schema. It describes how to declare the elements and attributes that appear in XML documents, the distinctions between simple and complex types, defining complex types, the use of simple types for element and attribute values, schema annotation, a simple mechanism for re-using element and attribute definitions, and nil values.

Section 3, the first advanced section in the primer, explains the basics of how namespaces are used in XML and schema documents. This section is important for understanding many of the topics that appear in the other advanced sections.

Section 4, the second advanced section in the primer, describes mechanisms for deriving types from existing types, and for controlling these derivations. The section also describes mechanisms for merging together fragments of a schema from multiple sources, and for element substitution.

Section 5 covers more advanced features, including a mechanism for specifying uniqueness among attributes and elements, a mechanism for using types across namespaces, a mechanism for extending types based on namespaces, and a description of how documents are checked for conformance.

In addition to the sections just described, the primer contains a number of appendices that provide detailed reference information on simple types and a regular expression language.

The primer is a non-normative document, which means that it does not provide a definitive (from the W3C's point of view) specification of the XML Schema language. The examples and other explanatory material in this document are provided to help you understand XML Schema, but they may not always provide definitive answers. In such cases, you will need to refer to the XML Schema specification, and to help you do this, we provide many links pointing to the relevant parts of the specification. More specifically, XML Schema items mentioned in the primer text are linked to an index of element names and attributes, and a summary table of datatypes, both in the primer. The table and the index contain links to the relevant sections of XML Schema parts 1 and 2.
2 Basic Concepts: The Purchase Order

The purpose of a schema is to define a class of XML documents, and so the term "instance document" is often used to describe an XML document that conforms to a particular schema. In fact, neither instances nor schemas need to exist as documents per se — they may exist as streams of bytes sent between applications, as fields in a database record, or as collections of XML Infoset "Information Items" — but to simplify the primer, we have chosen to always refer to instances and schemas as if they are documents and files.

Let us start by considering an instance document in a file called po.xml. It describes a purchase order generated by a home products ordering and billing application:

The Purchase Order, po.xml

```xml
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

The purchase order consists of a main element, `purchaseOrder`, and the subelements `shipTo`, `billTo`, `comment`, and `items`. These subelements (except `comment`) in turn contain other subelements, and so on, until a subelement such as `USPrice` contains a number rather than any subelements. Elements that contain subelements or carry attributes are said to have complex types, whereas elements that contain numbers (and strings, and dates, etc.) but do not contain any subelements are said to have simple types. Some elements have attributes; attributes always have simple types.

The complex types in the instance document, and some of the simple types, are defined in the schema for purchase orders. The other simple types are defined as part of XML Schema's repertoire of built-in simple types.
Before going on to examine the purchase order schema, we digress briefly to mention the association between the instance document and the purchase order schema. As you can see by inspecting the instance document, the purchase order schema is not mentioned. An instance is not actually required to reference a schema, and although many will, we have chosen to keep this first section simple, and to assume that any processor of the instance document can obtain the purchase order schema without any information from the instance document. In later sections, we will introduce explicit mechanisms for associating instances and schemas.

2.1 The Purchase Order Schema

The purchase order schema is contained in the file `po.xsd`:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items" type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
  </xsd:complexType>

  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
The purchase order schema consists of a `schema` element and a variety of subelements, most notably `element`, `complexType`, and `simpleType` which determine the appearance of elements and their content in instance documents.

Each of the elements in the schema has a prefix `xsd`: which is associated with the XML Schema namespace through the declaration, `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`, that appears in the `schema` element. The prefix `xsd:` is used by convention to denote the XML Schema namespace, although any prefix can be used. The same prefix, and hence the same association, also appears on the names of built-in simple types, e.g. `xsd:string`. The purpose of the association is to identify the elements and simple types as belonging to the vocabulary of the XML Schema language rather than the vocabulary of the schema author. For the sake of clarity in the text, we just mention the names of elements and simple types (e.g. `simpleType`), and omit the prefix.

### 2.2 Complex Type Definitions, Element & Attribute Declarations

In XML Schema, there is a basic difference between complex types which allow elements in their content and may carry attributes, and simple types which cannot have element content and cannot carry attributes. There is also a major distinction between definitions which create new types (both simple and complex), and declarations which enable elements and attributes with specific names and types (both simple and complex) to appear in document instances. In this section, we focus on defining complex types and declaring the elements and attributes that appear within them.

New complex types are defined using the `complexType` element and such definitions typically contain a set of element declarations, element references, and attribute declarations. The declarations are not themselves types, but rather an association between a name and the constraints which govern the appearance of that name in documents governed by the associated schema. Elements are declared using the `element` element, and attributes are declared using the `attribute` element. For example, `USAddress` is defined as a complex type, and within the definition of `USAddress` we see five element declarations and one attribute declaration:

Defining the `USAddress` Type

```xml
<xsd:complexType name="USAddress"/>
```
The consequence of this definition is that any element appearing in an instance whose type is declared to be USAddress (e.g. shipTo in po.xml) must consist of five elements and one attribute. These elements must be called name, street, city, state and zip as specified by the values of the declarations' name attributes, and the elements must appear in the same sequence (order) in which they are declared. The first four of these elements will each contain a string, and the fifth will contain a number. The element whose type is declared to be USAddress may appear with an attribute called country which must contain the string US.

The USAddress definition contains only declarations involving the simple types: string, decimal and NMTOKEN. In contrast, the PurchaseOrderType definition contains element declarations involving complex types, e.g. USAddress, although note that both declarations use the same type attribute to identify the type, regardless of whether the type is simple or complex.

Defining PurchaseOrderType

    <xsd:complexType name="PurchaseOrderType">
        <xsd:sequence>
            <xsd:element name="shipTo" type="USAddress"/>
            <xsd:element name="billTo" type="USAddress"/>
            <xsd:element ref="comment" minOccurs="0"/>
            <xsd:element name="items" type="Items"/>
        </xsd:sequence>
        <xsd:attribute name="orderDate" type="xsd:date"/>
    </xsd:complexType>

In defining PurchaseOrderType, two of the element declarations, for shipTo and billTo, associate different element names with the same complex type, namely USAddress. The consequence of this definition is that any element appearing in an instance document (e.g. po.xml) whose type is declared to be PurchaseOrderType must consist of elements named shipTo and billTo, each containing the five subelements (name, street, city, state and zip) that were declared as part of USAddress. The shipTo and billTo elements may also carry the country attribute that was declared as part of USAddress.

The PurchaseOrderType definition contains an orderDate attribute declaration which, like the country attribute declaration, identifies a simple type. In fact, all attribute declarations must reference simple types because, unlike element declarations, attributes cannot contain other elements or other attributes.

The element declarations we have described so far have each associated a name with an existing type definition. Sometimes it is preferable to use an existing element rather than declare a new element, for example:

    <xsd:element ref="comment" minOccurs="0"/>
This declaration references an existing element, comment, that was declared elsewhere in the purchase order schema. In general, the value of the ref attribute must reference a global element, i.e. one that has been declared under schema rather than as part of a complex type definition. The consequence of this declaration is that an element called comment may appear in an instance document, and its content must be consistent with that element's type, in this case, string.

2.2.1 Occurrence Constraints

The comment element is optional within PurchaseOrderType because the value of the minOccurs attribute in its declaration is 0. In general, an element is required to appear when the value of minOccurs is 1 or more. The maximum number of times an element may appear is determined by the value of a maxOccurs attribute in its declaration. This value may be a positive integer such as 41, or the term unbounded to indicate there is no maximum number of occurrences. The default value for both the minOccurs and the maxOccurs attributes is 1. Thus, when an element such as comment is declared without a maxOccurs attribute, the element may not occur more than once. Be sure that if you specify a value for only the minOccurs attribute, it is less than or equal to the default value of maxOccurs, i.e. it is 0 or 1. Similarly, if you specify a value for only the maxOccurs attribute, it must be greater than or equal to the default value of minOccurs, i.e. 1 or more. If both attributes are omitted, the element must appear exactly once.

Attributes may appear once or not at all, but no other number of times, and so the syntax for specifying occurrences of attributes is different than the syntax for elements. In particular, attributes can be declared with a use attribute to indicate whether the attribute is required (see for example, the partNum attribute declaration in po.xsd), optional, or even prohibited.

Default values of both attributes and elements are declared using the default attribute, although this attribute has a slightly different consequence in each case. When an attribute is declared with a default value, the value of the attribute is whatever value appears as the attribute's value in an instance document; if the attribute does not appear in the instance document, the schema processor provides the attribute with a value equal to that of the default attribute. Note that default values for attributes only make sense if the attributes themselves are optional, and so it is an error to specify both a default value and anything other than a value of optional for use.

The schema processor treats defaulted elements slightly differently. When an element is declared with a default value, the value of the element is whatever value appears as the element's content in the instance document; if the element appears without any content, the schema processor provides the element with a value equal to that of the default attribute. However, if the element does not appear in the instance document, the schema processor does not provide the element at all. In summary, the differences between element and attribute defaults can be stated as: Default attribute values apply when attributes are missing, and default element values apply when elements are empty.

The fixed attribute is used in both attribute and element declarations to ensure that the attributes and elements are set to particular values. For example, po.xsd contains a declaration for the country attribute, which is declared with a fixed value US. This declaration means that the appearance of a country attribute in an instance document is optional (the default value of use is optional), although if the attribute does appear, its value must be US, and if the attribute does not appear, the schema processor will provide a country attribute with the value US. Note that the concepts of a fixed value and a default value are mutually exclusive, and so it is an error for a declaration to contain both fixed and default attributes.

The values of the attributes used in element and attribute declarations to constrain their occurrences are summarized in Table 1.
### Table 1. Occurrence Constraints for Elements and Attributes

<table>
<thead>
<tr>
<th>Elements</th>
<th>Attributes</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(minOccurs, maxOccurs)</td>
<td>use, fixed, default</td>
<td></td>
</tr>
<tr>
<td>fixed, default</td>
<td>required, −, −, −</td>
<td>element/attribute must appear once, it may have any value</td>
</tr>
<tr>
<td>(1, 1) −, −</td>
<td>required, −</td>
<td>element/attribute must appear once, its value must be 37</td>
</tr>
<tr>
<td>(1, 1) 37, −</td>
<td>required, 37, −</td>
<td>element must appear twice or more, its value must be 37; in general, minOccurs and maxOccurs values may be positive integers, and maxOccurs value may also be &quot;unbounded&quot;</td>
</tr>
<tr>
<td>(2, unbounded) 37, −</td>
<td>n/a</td>
<td>element must appear twice or more, its value must be 37; in general, minOccurs and maxOccurs values may be positive integers, and maxOccurs value may also be &quot;unbounded&quot;</td>
</tr>
<tr>
<td>(0, 1) −, −</td>
<td>optional, −, −</td>
<td>element/attribute may appear once, it may have any value</td>
</tr>
<tr>
<td>(0, 1) 37, −</td>
<td>optional, 37, −</td>
<td>element/attribute may appear once, if it does appear its value must be 37, if it does not appear its value is 37</td>
</tr>
<tr>
<td>(0, 1) −, 37</td>
<td>optional, −, 37</td>
<td>element/attribute may appear once; if it does not appear its value is 37, otherwise its value is that given</td>
</tr>
<tr>
<td>(0, 2) −, 37</td>
<td>n/a</td>
<td>element may appear once, twice, or not at all; if the element does not appear it is not provided; if it does appear and it is empty, its value is 37; otherwise its value is that given; in general, minOccurs and maxOccurs values may be positive integers, and maxOccurs value may also be &quot;unbounded&quot;</td>
</tr>
<tr>
<td>(0, 0) −, −</td>
<td>prohibited, −, −</td>
<td>element/attribute must not appear</td>
</tr>
</tbody>
</table>

Note that neither minOccurs, maxOccurs, nor use may appear in the declarations of global elements and attributes.

#### 2.2.2 Global Elements & Attributes

Global elements, and global attributes, are created by declarations that appear as the children of the schema element. Once declared, a global element or a global attribute can be referenced in one or more declarations using the ref attribute as described above. A declaration that references a global element enables the referenced element to appear in the instance document in the context of the referencing declaration. So, for example, the comment element appears in po.xml at the same level as the shipTo, billTo and items elements because the declaration that references comment appears in the complex type definition at the same level as the declarations of the other three elements.

The declaration of a global element also enables the element to appear at the top−level of an instance document. Hence purchaseOrder, which is declared as a global element in po.xsd, can appear as the top−level element in po.xml. Note that this rationale will also allow a comment element to appear as the top−level element in a document like po.xml.

There are a number of caveats concerning the use of global elements and attributes. One caveat is that global declarations cannot contain references; global declarations must identify simple and complex types directly. Put concretely, global declarations cannot contain the ref attribute, they must use the type attribute (or, as we describe shortly, be followed by an anonymous type definition). A second caveat is that cardinality
constraints cannot be placed on global declarations, although they can be placed on local declarations that reference global declarations. In other words, global declarations cannot contain the attributes minOccurs, maxOccurs, or use.

### 2.2.3 Naming Conflicts

We have now described how to define new complex types (e.g. PurchaseOrderType), declare elements (e.g. purchaseOrder) and declare attributes (e.g. orderDate). These activities generally involve naming, and so the question naturally arises: What happens if we give two things the same name? The answer depends upon the two things in question, although in general the more similar are the two things, the more likely there will be a conflict.

Here are some examples to illustrate when same names cause problems. If the two things are both types, say we define a complex type called USStates and a simple type called USStates, there is a conflict. If the two things are a type and an element or attribute, say we define a complex type called USAddress and we declare an element called USAddress, there is no conflict. If the two things are elements within different types (i.e. not global elements), say we declare one element called name as part of the USAddress type and a second element called name as part of the Item type, there is no conflict. (Such elements are sometimes called local element declarations.) Finally, if the two things are both types and you define one and XML Schema has defined the other, say you define a simple type called decimal, there is no conflict. The reason for the apparent contradiction in the last example is that the two types belong to different namespaces. We explore the use of namespaces in schema in a later section.

### 2.3 Simple Types

The purchase order schema declares several elements and attributes that have simple types. Some of these simple types, such as string and decimal, are built in to XML Schema, while others are derived from the built−in's. For example, the partNum attribute has a type called SKU (Stock Keeping Unit) that is derived from string. Both built−in simple types and their derivations can be used in all element and attribute declarations. Table 2 lists all the simple types built in to XML Schema, along with examples of the different types.

<table>
<thead>
<tr>
<th>Simple Type</th>
<th>Examples (delimited by commas)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Confirm this is electric</td>
<td></td>
</tr>
<tr>
<td>normalizedString</td>
<td>Confirms this is electric</td>
<td>see (3)</td>
</tr>
<tr>
<td>token</td>
<td>Confirm this is electric</td>
<td>see (4)</td>
</tr>
<tr>
<td>byte</td>
<td>−1, 126</td>
<td>see (2)</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>0, 126</td>
<td>see (2)</td>
</tr>
<tr>
<td>base64Binary</td>
<td>GpM7</td>
<td></td>
</tr>
<tr>
<td>hexBinary</td>
<td>0FB7</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>−126789, −1, 0, 1, 126789</td>
<td>see (2)</td>
</tr>
<tr>
<td>positiveInteger</td>
<td>1, 126789</td>
<td>see (2)</td>
</tr>
<tr>
<td>negativeInteger</td>
<td>−126789, −1</td>
<td>see (2)</td>
</tr>
<tr>
<td>Type</td>
<td>Values</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>0, 1, 126789</td>
<td>see (2)</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>−126789, −1, 0</td>
<td>see (2)</td>
</tr>
<tr>
<td>int</td>
<td>−1, 126789675</td>
<td>see (2)</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>0, 1267896754</td>
<td>see (2)</td>
</tr>
<tr>
<td>long</td>
<td>−1, 12678967543233</td>
<td>see (2)</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>0, 12678967543233</td>
<td>see (2)</td>
</tr>
<tr>
<td>short</td>
<td>−1, 12678</td>
<td>see (2)</td>
</tr>
<tr>
<td>unsignedShort</td>
<td>0, 12678</td>
<td>see (2)</td>
</tr>
<tr>
<td>decimal</td>
<td>−1.23, 0, 123.4, 1000.00</td>
<td>equivalent to single-precision 32-bit floating point, NaN is &quot;not a number&quot;, see (2)</td>
</tr>
<tr>
<td>float</td>
<td>−INF, −1E4, −0, 0, 12.78E−2, 12, INF, NaN</td>
<td>equivalent to single-precision 32-bit floating point, NaN is &quot;not a number&quot;, see (2)</td>
</tr>
<tr>
<td>double</td>
<td>−INF, −1E4, −0, 0, 12.78E−2, 12, INF, NaN</td>
<td>equivalent to double-precision 64-bit floating point, see (2)</td>
</tr>
<tr>
<td>boolean</td>
<td>true, false, 1, 0</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>13:20:00.000, 13:20:00.000−05:00</td>
<td>May 31st 1999 at 1.20pm Eastern Standard Time which is 5 hours behind Co-Ordinated Universal Time, see (2)</td>
</tr>
<tr>
<td>dateTime</td>
<td>1999−05−31T13:20:00.000−05:00</td>
<td>May 31st 1999 at 1.20pm Eastern Standard Time which is 5 hours behind Co-Ordinated Universal Time, see (2)</td>
</tr>
<tr>
<td>duration</td>
<td>P1Y2M3DT10H30M12.3S</td>
<td>1 year, 2 months, 3 days, 10 hours, 30 minutes, and 12.3 seconds</td>
</tr>
<tr>
<td>date</td>
<td>1999−05−31</td>
<td>see (2)</td>
</tr>
<tr>
<td>gMonth</td>
<td>---05---</td>
<td>May, see (2) (5)</td>
</tr>
<tr>
<td>gYear</td>
<td>1999</td>
<td>1999, see (2) (5)</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>1999−02</td>
<td>the month of February 1999, regardless of the number of days, see (2) (5)</td>
</tr>
<tr>
<td>gDay</td>
<td>---31</td>
<td>the 31st day, see (2) (5)</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>---05−31</td>
<td>every May 31st, see (2) (5)</td>
</tr>
<tr>
<td>Name</td>
<td>shipTo</td>
<td>XML 1.0 Name type</td>
</tr>
<tr>
<td>QName</td>
<td>po:USAddress</td>
<td>XML Namespace QName</td>
</tr>
<tr>
<td>NCName</td>
<td>USAddress</td>
<td>XML Namespace NCName, i.e. a QName without the prefix and colon</td>
</tr>
<tr>
<td>anyURI</td>
<td><a href="http://www.example.com/">http://www.example.com/</a></td>
<td></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.example.com/doc.html#ID5">http://www.example.com/doc.html#ID5</a></td>
<td></td>
</tr>
<tr>
<td>language</td>
<td>en–GB, en–US, fr</td>
<td>valid values for xml:lang as defined in XML 1.0</td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td>XML 1.0 ID attribute type, see (1)</td>
</tr>
</tbody>
</table>

2.2 Complex Type Definitions, Element & Attribute Declarations

11
<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDREF</td>
<td>XML 1.0 IDREF attribute type, see (1)</td>
</tr>
<tr>
<td>IDREFS</td>
<td>XML 1.0 IDREFS attribute type, see (1)</td>
</tr>
<tr>
<td>ENTITY</td>
<td>XML 1.0 ENTITY attribute type, see (1)</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>XML 1.0 ENTITIES attribute type, see (1)</td>
</tr>
<tr>
<td>NOTATION</td>
<td>XML 1.0 NOTATION attribute type, see (1)</td>
</tr>
<tr>
<td>NMTOKEN</td>
<td>US, Brésil</td>
</tr>
<tr>
<td>NMTOKENS</td>
<td>US UK, Brésil Canada Mexique</td>
</tr>
</tbody>
</table>

Notes: (1) To retain compatibility between XML Schema and XML 1.0 DTDs, the simple types ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS should only be used in attributes. (2) A value of this type can be represented by more than one lexical format, e.g. 100 and 1.0E2 are both valid float formats representing "one hundred". However, rules have been established for this type that define a canonical lexical format, see XML Schema Part 2. (3) Newline, tab and carriage-return characters in a normalizedString type are converted to space characters before schema processing. (4) As normalizedString, and adjacent space characters are collapsed to a single space character, and leading and trailing spaces are removed. (5) The 'g' prefix signals time periods in the Gregorian calendar.

New simple types are defined by deriving them from existing simple types (built-in's and derived). In particular, we can derive a new simple type by restricting an existing simple type, in other words, the legal range of values for the new type are a subset of the existing type's range of values. We use the `<xsd:simpleType>` element to define and name the new simple type. We use the `<xsd:restriction>` element to indicate the existing (base) type, and to identify the "facets" that constrain the range of values. A complete list of facets is provided in Appendix B.

Suppose we wish to create a new type of integer called `myInteger` whose range of values is between 10000 and 99999 (inclusive). We base our definition on the built-in simple type `integer`, whose range of values also includes integers less than 10000 and greater than 99999. To define `myInteger`, we restrict the range of the `integer` base type by employing two facets called `minInclusive` and `maxInclusive`:

```
<xsdsimpleType name="myInteger">
  <xsd:restriction base="xsds:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

The example shows one particular combination of a base type and two facets used to define `myInteger`, but a look at the list of built-in simple types and their facets (Appendix B) should suggest other viable combinations.

The purchase order schema contains another, more elaborate, example of a simple type definition. A new simple type called SKU is derived (by restriction) from the simple type `string`. Furthermore, we constrain the values of SKU using a facet called `pattern` in conjunction with the regular expression "\d{3}-[A-Z]{2}" that is read "three digits followed by a hyphen followed by two upper-case ASCII letters":

```
<xsdsimpleType name="SKU">
  <xsd:restriction base="xsds:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```
This regular expression language is described more fully in Appendix D.

XML Schema defines fifteen facets which are listed in Appendix B. Among these, the **enumeration** facet is particularly useful and it can be used to constrain the values of almost every simple type, except the **boolean** type. The **enumeration** facet limits a simple type to a set of distinct values. For example, we can use the **enumeration** facet to define a new simple type called **USState**, derived from **string**, whose value must be one of the standard US state abbreviations:

Using the **Enumeration** Facet

```xml
<xsd:simpleType name="USState">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="AK"/>
      <xsd:enumeration value="AL"/>
      <xsd:enumeration value="AR"/>
      <!-- and so on ... -->
   </xsd:restriction>
</xsd:simpleType>
```

**USState** would be a good replacement for the **string** type currently used in the **state** element declaration. By making this replacement, the legal values of a **state** element, i.e. the **state** subelements of **billTo** and **shipTo**, would be limited to one of AK, AL, AR, etc. Note that the enumeration values specified for a particular type must be unique.

### 2.3.1 List Types

XML Schema has the concept of a list type, in addition to the so–called atomic types that constitute most of the types listed in Table 2. (Atomic types, list types, and the union types described in the next section are collectively called simple types.) The value of an atomic type is indivisible from XML Schema's perspective. For example, the **NMTOKEN** value **US** is indivisible in the sense that no part of **US**, such as the character "S", has any meaning by itself. In contrast, list types are comprised of sequences of atomic types and consequently the parts of a sequence (the "atoms") themselves are meaningful. For example, **NMTOKENS** is a list type, and an element of this type would be a white–space delimited list of **NMTOKEN's**, such as "US UK FR". XML Schema has three built–in list types, they are **NMTOKENS**, **IDREFS**, and **ENTITIES**.

In addition to using the built–in list types, you can create new list types by derivation from existing atomic types. (You cannot create list types from existing list types, nor from complex types.) For example, to create a list of **myInteger's**:

Creating a List of **myInteger's**

```xml
<xsd:simpleType name="listOfMyIntType">
   <xsd:list itemType="myInteger"/>
</xsd:simpleType>
```

And an element in an instance document whose content conforms to **listOfMyIntType** is:

```xml
<listOfMyInt>20003 15037 95977 95945</listOfMyInt>
```
Several facets can be applied to list types: `length`, `minLength`, `maxLength`, and `enumeration`. For example, to define a list of exactly six US states (`SixUSStates`), we first define a new list type called `USStateList` from `USState`, and then we derive `SixUSStates` by restricting `USStateList` to only six items:

**List Type for Six US States**

```xml
<xsd:simpleType name="USStateList">
  <xsd:list itemType="USState"/>
</xsd:simpleType>

<xsd:simpleType name="SixUSStates">
  <xsd:restriction base="USStateList">
    <xsd:length value="6"/>
  </xsd:restriction>
</xsd:simpleType>
```

Elements whose type is `SixUSStates` must have six items, and each of the six items must be one of the (atomic) values of the enumerated type `USState`, for example:

```xml
<sixStates>PA NY CA NY LA AK</sixStates>
```

Note that it is possible to derive a list type from the atomic type `string`. However, a `string` may contain white space, and white space delimits the items in a list type, so you should be careful using list types whose base type is `string`. For example, suppose we have defined a list type with a `length` facet equal to 3, and base type `string`, then the following 3 item list is legal:

```xml
Asie Europe Afrique
```

But the following 3 "item" list is illegal:

```xml
Asie Europe Amérique Latine
```

Even though "Amérique Latine" may exist as a single string outside of the list, when it is included in the list, the whitespace between Amérique and Latine effectively creates a fourth item, and so the latter example will not conform to the 3–item list type.

### 2.3.2 Union Types

Atomic types and list types enable an element or an attribute value to be one or more instances of one atomic type. In contrast, a union type enables an element or attribute value to be one or more instances of one type drawn from the union of multiple atomic and list types. To illustrate, we create a union type for representing American states as singleton letter abbreviations or lists of numeric codes. The `zipUnion` union type is built from one atomic type and one list type:

**Union Type for Zipcodes**

```xml
<xsd:simpleType name="zipUnion">
  <xsd:union memberTypes="USState listOfMyIntType"/>
</xsd:simpleType>
```

When we define a union type, the `memberTypes` attribute value is a list of all the types in the union.
Now, assuming we have declared an element called \texttt{zips} of type \texttt{zipUnion}, valid instances of the element are:

\begin{verbatim}
<zips>CA</zips>
<zips>95630 95977 95945</zips>
<zips>AK</zips>
\end{verbatim}

Two facets, \texttt{pattern} and \texttt{enumeration}, can be applied to a union type.

### 2.4 Anonymous Type Definitions

Schemas can be constructed by defining sets of named types such as \texttt{PurchaseOrderType} and then declaring elements such as \texttt{purchaseOrder} that reference the types using the \texttt{type=} construction. This style of schema construction is straightforward but it can be unwieldy, especially if you define many types that are referenced only once and contain very few constraints. In these cases, a type can be more succinctly defined as an anonymous type which saves the overhead of having to be named and explicitly referenced.

The definition of the type \texttt{Items} in \texttt{po.xsd} contains two element declarations that use anonymous types (\texttt{item} and \texttt{quantity}). In general, you can identify anonymous types by the lack of a \texttt{type=} in an element (or attribute) declaration, and by the presence of an un–named (simple or complex) type definition:

Two Anonymous Type Definitions

\begin{verbatim}
<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice" type="xsd:decimal"/>
          <xsd:element ref="comment" minOccurs="0"/>
          <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="partNum" type="SKU" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
\end{verbatim}

In the case of the \texttt{item} element, it has an anonymous complex type consisting of the elements \texttt{productName}, \texttt{quantity}, \texttt{USPrice}, \texttt{comment}, and \texttt{shipDate}, and an attribute called \texttt{partNum}. In the case of the \texttt{quantity} element, it has an anonymous simple type derived from \texttt{integer} whose value ranges between 1 and 99.
2.5 Element Content

The purchase order schema has many examples of elements containing other elements (e.g. items), elements having attributes and containing other elements (e.g. shipTo), and elements containing only a simple type of value (e.g. USPrice). However, we have not seen an element having attributes but containing only a simple type of value, nor have we seen an element that contains other elements mixed with character content, nor have we seen an element that has no content at all. In this section we'll examine these variations in the content models of elements.

2.5.1 Complex Types from Simple Types

Let us first consider how to declare an element that has an attribute and contains a simple value. In an instance document, such an element might appear as:

```xml
<internationalPrice currency="EUR">423.46</internationalPrice>
```

The purchase order schema declares a USPrice element that is a starting point:

```xml
<xsd:element name="USPrice" type="decimal"/>
```

Now, how do we add an attribute to this element? As we have said before, simple types cannot have attributes, and \_decimal is a simple type. Therefore, we must define a complex type to carry the attribute declaration. We also want the content to be simple type \_decimal. So our original question becomes: How do we define a complex type that is based on the simple type \_decimal? The answer is to derive a new complex type from the simple type \_decimal:

Deriving a Complex Type from a Simple Type

```xml
<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="currency" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

We use the \_complexType element to start the definition of a new (anonymous) type. To indicate that the content model of the new type contains only character data and no elements, we use a \_simpleContent element. Finally, we derive the new type by extending the simple \_decimal type. The extension consists of adding a currency attribute using a standard attribute declaration. (We cover type derivation in detail in Section 4.) The \_internationalPrice element declared in this way will appear in an instance as shown in the example at the beginning of this section.

2.5.2 Mixed Content

The construction of the purchase order schema may be characterized as elements containing subelements, and the deepest subelements contain character data. XML Schema also provides for the construction of schemas where character data can appear alongside subelements, and character data is not confined to the deepest subelements.
To illustrate, consider the following snippet from a customer letter that uses some of the same elements as the purchase order:

Snippet of Customer Letter

<letterBody>
<salutation>Dear Mr.<name>Robert Smith</name>.</salutation>
Your order of <quantity>1</quantity> <productName>Baby Monitor</productName> shipped from our warehouse on <shipDate>1999−05−21</shipDate>. ....
</letterBody>

Notice the text appearing between elements and their child elements. Specifically, text appears between the elements \texttt{salutation}, \texttt{quantity}, \texttt{productName} and \texttt{shipDate} which are all children of \texttt{letterBody}, and text appears around the element name which is the child of a child of \texttt{letterBody}. The following snippet of a schema declares \texttt{letterBody}:

Snippet of Schema for Customer Letter

<xsd:element name="letterBody">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element name="salutation">
        <xsd:complexType mixed="true">
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="quantity" type="xsd:positiveInteger"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

The elements appearing in the customer letter are declared, and their types are defined using the \texttt{element} and \texttt{complexType} element constructions we have seen before. To enable character data to appear between the child–elements of \texttt{letterBody}, the \texttt{mixed} attribute on the type definition is set to true.

Note that the \texttt{mixed} model in XML Schema differs fundamentally from the \texttt{mixed} model in XML 1.0. Under the XML Schema mixed model, the order and number of child elements appearing in an instance must agree with the order and number of child elements specified in the model. In contrast, under the XML 1.0 mixed model, the order and number of child elements appearing in an instance cannot be constrained. In summary, XML Schema provides full validation of mixed models in contrast to the partial schema validation provided by XML 1.0.

2.5.3 Empty Content

Now suppose that we want the \texttt{internationalPrice} element to convey both the unit of currency and the price as attribute values rather than as separate attribute and content values. For example:

<internationalPrice currency="EUR" value="423.46"/>
Such an element has no content at all; its content model is empty. To define a type whose content is empty, we essentially define a type that allows only elements in its content, but we do not actually declare any elements and so the type's content model is empty:

An Empty Complex Type

```xml
<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
        <xsd:attribute name="currency" type="xsd:string"/>
        <xsd:attribute name="value" type="xsd:decimal"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

In this example, we define an (anonymous) type having `complexContent`, i.e. only elements. The `complexContent` element signals that we intend to restrict or extend the content model of a complex type, and the `restriction` of `anyType` declares two attributes but does not introduce any element content (see Section 4.4 for more details on restriction). The `internationalPrice` element declared in this way may legitimately appear in an instance as shown in the example above.

The preceding syntax for an empty−content element is relatively verbose, and it is possible to declare the `internationalPrice` element more compactly:

Shorthand for an Empty Complex Type

```xml
<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:attribute name="currency" type="xsd:string"/>
    <xsd:attribute name="value" type="xsd:decimal"/>
  </xsd:complexType>
</xsd:element>
```

This compact syntax works because a complex type defined without any `simpleContent` or `complexContent` is interpreted as shorthand for complex content that restricts `anyType`.

2.5.4 anyType

The `anyType` represents an abstraction called the _ur−type_ which is the base type from which all simple and complex types are derived. An `anyType` type does not constrain its content in any way. It is possible to use `anyType` like other types, for example:

```xml
<xsd:element name="anything" type="xsd:anyType"/>
```

The content of the element declared in this way is unconstrained, so the element value may be 423.46, but it may be any other sequence of characters as well, or indeed a mixture of characters and elements. In fact, `anyType` is the default type when none is specified, so the above could also be written as follows:

```xml
<xsd:element name="anything"/>
```

If unconstrained element content is needed, for example in the case of elements containing prose which requires embedded markup to support internationalization, then the default declaration or a slightly restricted
form of it may be suitable. The text type described in Section 5.5 is an example of such a type that is suitable for such purposes.

### 2.6 Annotations

XML Schema provides three elements for annotating schemas for the benefit of both human readers and applications. In the purchase order schema, we put a basic schema description and copyright information inside the `documentation` element, which is the recommended location for human readable material. We recommend you use the `xml:lang` attribute with any `documentation` elements to indicate the language of the information. Alternatively, you may indicate the language of all information in a schema by placing an `xml:lang` attribute on the schema element.

The `appInfo` element, which we did not use in the purchase order schema, can be used to provide information for tools, stylesheets and other applications. An interesting example using `appInfo` is a schema that describes the simple types in XML Schema Part 2: Datatypes. Information describing this schema, e.g. which facets are applicable to particular simple types, is represented inside `appInfo` elements, and this information was used by an application to automatically generate text for the XML Schema Part 2 document.

Both `documentation` and `appInfo` appear as subelements of `annotation`, which may itself appear at the beginning of most schema constructions. To illustrate, the following example shows `annotation` elements appearing at the beginning of an element declaration and a complex type definition:

Annotating an Element Declaration & Complex Type Definition

```xml
<xsd:element name="internationalPrice">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      element declared with anonymous type
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        empty anonymous type with 2 attributes
      </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
        <xsd:attribute name="currency" type="xsd:string"/>
        <xsd:attribute name="value" type="xsd:decimal"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

The `annotation` element may also appear at the beginning of other schema constructions such as those indicated by the elements `schema`, `simpleType`, and `attribute`.

### 2.7 Building Content Models

The definitions of complex types in the purchase order schema all declare sequences of elements that must appear in the instance document. The occurrence of individual elements declared in the so-called content models of these types may be optional, as indicated by a 0 value for the attribute `minOccurs` (e.g. in
comment), or be otherwise constrained depending upon the values of minOccurs and maxOccurs. XML Schema also provides constraints that apply to groups of elements appearing in a content model. These constraints mirror those available in XML 1.0 plus some additional constraints. Note that the constraints do not apply to attributes.

XML Schema enables groups of elements to be defined and named, so that the elements can be used to build up the content models of complex types (thus mimicking common usage of parameter entities in XML 1.0). Un–named groups of elements can also be defined, and along with elements in named groups, they can be constrained to appear in the same order (sequence) as they are declared. Alternatively, they can be constrained so that only one of the elements may appear in an instance.

To illustrate, we introduce two groups into the PurchaseOrderType definition from the purchase order schema so that purchase orders may contain either separate shipping and billing addresses, or a single address for those cases in which the shippee and billee are co–located:

Nested Choice and Sequence Groups

```xml
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:group ref="shipAndBill"/>
      <xsd:element name="singleUSAddress" type="USAddress"/>
    </xsd:choice>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:group name="shipAndBill">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
  </xsd:sequence>
</xsd:group>
```

The choice group element allows only one of its children to appear in an instance. One child is an inner group element that references the named group shipAndBill consisting of the element sequence shipTo, billTo, and the second child is a singleUSAddress. Hence, in an instance document, the purchaseOrder element must contain either a shipTo element followed by a billTo element or a singleUSAddress element. The choice group is followed by the comment and items element declarations, and both the choice group and the element declarations are children of a sequence group. The effect of these various groups is that the address element(s) must be followed by comment and items elements in that order.

There exists a third option for constraining elements in a group: All the elements in the group may appear once or not at all, and they may appear in any order. The all group (which provides a simplified version of the SGML &−Connector) is limited to the top–level of any content model. Moreover, the group's children must all be individual elements (no groups), and no element in the content model may appear more than once, i.e. the permissible values of minOccurs and maxOccurs are 0 and 1. For example, to allow the child elements of purchaseOrder to appear in any order, we could redefine PurchaseOrderType as:

An 'All' Group

2.6 Annotations
By this definition, a comment element may optionally appear within purchaseOrder, and it may appear before or after any shipTo, billTo and items elements, but it can appear only once. Moreover, the stipulations of an all group do not allow us to declare an element such as comment outside the group as a means of enabling it to appear more than once. XML Schema stipulates that an all group must appear as the sole child at the top of a content model. In other words, the following is illegal:

Illegal Example with an 'All' Group

Finally, named and un-named groups that appear in content models (represented by group and choice, sequence, all respectively) may carry minOccurs and maxOccurs attributes. By combining and nesting the various groups provided by XML Schema, and by setting the values of minOccurs and maxOccurs, it is possible to represent any content model expressible with an XML 1.0 DTD. Furthermore, the all group provides additional expressive power.

2.8 Attribute Groups

Suppose we want to provide more information about each item in a purchase order, for example, each item's weight and preferred shipping method. We can accomplish this by adding weightKg and shipBy attribute declarations to the item element's (anonymous) type definition:

Adding Attributes to the Inline Type Definition
Alternatively, we can create a named attribute group containing all the desired attributes of an item element, and reference this group by name in the item element declaration:

Adding Attributes Using an Attribute Group

```xml
<xs:element name="item" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="productName" type="xsd:string"/>
      <xs:element name="quantity">
        <xs:simpleType>
          <xs:restriction base="xsd:positiveInteger">
            <xs:maxExclusive value="100"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="USPrice" type="xsd:decimal"/>
      <xs:element ref="comment" minOccurs="0"/>
      <xs:element name="shipDate" type="xsd:date" minOccurs="0"/>
    </xs:sequence>
    <xs:attributeGroup ref="ItemDelivery"/>
  </xs:complexType>
</xs:element>
```

<!−− attributeGroup replaces individual declarations -->
<xs:attributeGroup ref="ItemDelivery"/>
</xs:complexType>
</xs:element>

```
<xs:attributeGroup name="ItemDelivery">
  <xs:attribute name="partNum" type="SKU" use="required"/>
  <xs:attribute name="weightKg" type="xsd:decimal"/>
  <xs:attribute name="shipBy">
    <xs:simpleType>
      <xs:restriction base="xsd:string">
        <xs:enumeration value="air"/>
        <xs:enumeration value="land"/>
        <xs:enumeration value="any"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:attributeGroup>
```
Using an attribute group in this way can improve the readability of schemas, and facilitates updating schemas because an attribute group can be defined and edited in one place and referenced in multiple definitions and declarations. These characteristics of attribute groups make them similar to parameter entities in XML 1.0. Note that an attribute group may contain other attribute groups. Note also that both attribute declarations and attribute group references must appear at the end of complex type definitions.

2.9 Nil Values

One of the purchase order items listed in po.xml, the Lawnmower, does not have a shipDate element. Within the context of our scenario, the schema author may have intended such absences to indicate items not yet shipped. But in general, the absence of an element does not have any particular meaning: It may indicate that the information is unknown, or not applicable, or the element may be absent for some other reason. Sometimes it is desirable to represent an unshipped item, unknown information, or inapplicable information explicitly with an element, rather than by an absent element. For example, it may be desirable to represent a "null" value being sent to or from a relational database with an element that is present. Such cases can be represented using XML Schema's nil mechanism which enables an element to appear with or without a non-nil value.

XML Schema's nil mechanism involves an "out of band" nil signal. In other words, there is no actual nil value that appears as element content, instead there is an attribute to indicate that the element content is nil. To illustrate, we modify the shipDate element declaration so that nils can be signalled:

```xml
<xsd:element name="shipDate" type="xsd:date" nillable="true"/>
```

And to explicitly represent that shipDate has a nil value in the instance document, we set the nil attribute (from the XML Schema namespace for instances) to true:

```xml
<shipDate xsi:nil="true"></shipDate>
```

The nil attribute is defined as part of the XML Schema namespace for instances, http://www.w3.org/2001/XMLSchema-instance, and so it must appear in the instance document with a prefix (such as xsi:) associated with that namespace. (As with the xsd: prefix, the xsi: prefix is used by convention only.) Note that the nil mechanism applies only to element values, and not to attribute values. An element with xsi:nil="true" may not have any element content but it may still carry attributes.

3. Advanced Concepts I: Namespaces, Schemas & Qualification

A schema can be viewed as a collection (vocabulary) of type definitions and element declarations whose names belong to a particular namespace called a target namespace. Target namespaces enable us to distinguish between definitions and declarations from different vocabularies. For example, target namespaces would enable us to distinguish between the declaration for element in the XML Schema language vocabulary, and a declaration for element in a hypothetical chemistry language vocabulary. The former is part of the http://www.w3.org/2001/XMLSchema target namespace, and the latter is part of another target namespace.

When we want to check that an instance document conforms to one or more schemas (through a process called schema validation), we need to identify which element and attribute declarations and type definitions in the schemas should be used to check which elements and attributes in the instance document. The target namespace plays an important role in the identification process. We examine the role of the target namespace...
in the next section.

The schema author also has several options that affect how the identities of elements and attributes are represented in instance documents. More specifically, the author can decide whether or not the appearance of locally declared elements and attributes in an instance must be qualified by a namespace, using either an explicit prefix or implicitly by default. The schema author's choice regarding qualification of local elements and attributes has a number of implications regarding the structures of schemas and instance documents, and we examine some of these implications in the following sections.

3.1 Target Namespaces & Unqualified Locals

In a new version of the purchase order schema, po1.xsd, we explicitly declare a target namespace, and specify that both locally defined elements and locally defined attributes must be unqualified. The target namespace in po1.xsd is http://www.example.com/PO1, as indicated by the value of the targetNamespace attribute.

Qualification of local elements and attributes can be globally specified by a pair of attributes, elementFormDefault and attributeFormDefault, on the schema element, or can be specified separately for each local declaration using the form attribute. All such attributes' values may each be set to unqualified or qualified, to indicate whether or not locally declared elements and attributes must be unqualified.

In po1.xsd we globally specify the qualification of elements and attributes by setting the values of both elementFormDefault and attributeFormDefault to unqualified. Strictly speaking, these settings are unnecessary because the values are the defaults for the two attributes; we make them here to highlight the contrast between this case and other cases we describe later.

Purchase Order Schema with Target Namespace, po1.xsd

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:po="http://www.example.com/PO1"
    targetNamespace="http://www.example.com/PO1"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">

    <element name="purchaseOrder" type="po:PurchaseOrderType"/>
    <element name="comment" type="string"/>

    <complexType name="PurchaseOrderType">
        <sequence>
            <element name="shipTo" type="po:USAddress"/>
            <element name="billTo" type="po:USAddress"/>
            <element ref="po:comment" minOccurs="0"/>
            <!-- etc. -->
        </sequence>
        <!-- etc. -->
    </complexType>

    <complexType name="USAddress">
        <sequence>
            <element name="name" type="string"/>
            <element name="street" type="string"/>
            <!-- etc. -->
        </sequence>
        <!-- etc. -->
    </complexType>
</schema>
```
To see how the target namespace of this schema is populated, we examine in turn each of the type definitions and element declarations. Starting from the end of the schema, we first define a type called USAddress that consists of the elements name, street, etc. One consequence of this type definition is that the USAddress type is included in the schema's target namespace. We next define a type called PurchaseOrderType that consists of the elements shipTo, billTo, comment, etc. PurchaseOrderType is also included in the schema's target namespace. Notice that the type references in the three element declarations are prefixed, i.e. po:USAddress, po:USAddress and po:comment, and the prefix is associated with the namespace http://www.example.com/PO1. This is the same namespace as the schema's target namespace, and so a processor of this schema will know to look within this schema for the definition of the type USAddress and the declaration of the element comment. It is also possible to refer to types in another schema with a different target namespace, hence enabling re-use of definitions and declarations between schemas.

At the beginning of the schema po1.xsd, we declare the elements purchaseOrder and comment. They are included in the schema's target namespace. The purchaseOrder element's type is prefixed, for the same reason that USAddress is prefixed. In contrast, the comment element's type, string, is not prefixed. The po1.xsd schema contains a default namespace declaration, and so unprefixed types such as string and unprefixed elements such as _element and complexType are associated with the default namespace http://www.w3.org/2001/XMLSchema. In fact, this is the target namespace of XML Schema itself, and so a processor of po1.xsd will know to look within the schema of XML Schema --- otherwise known as the "schema for schemas" --- for the definition of the type _string and the declaration of the element called _element.

Let us now examine how the target namespace of the schema affects a conforming instance document:

A Purchase Order with Unqualified Locals, po1.xml

```xml
<?xml version="1.0"?>
<apo:purchaseOrder xmlns:apo="http://www.example.com/PO1"
    orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <!-- etc. -->
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <!-- etc. -->
    </billTo>
    <apo:comment>Hurry, my lawn is going wild!</apo:comment>
    <!-- etc. -->
</apo:purchaseOrder>
```

The instance document declares one namespace, http://www.example.com/PO1, and associates it with the prefix apo:. This prefix is used to qualify two elements in the document, namely purchaseOrder and comment. The namespace is the same as the target namespace of the schema in po1.xsd, and so a processor of the instance document will know to look in that schema for the declarations of purchaseOrder and comment. In fact, target namespaces are so named because of the sense in which
there exists a target namespace for the elements purchaseOrder and comment. Target namespaces in the
schema therefore control the validation of corresponding namespaces in the instance.

The prefix apo: is applied to the global elements purchaseOrder and comment elements.
Furthermore, elementFormDefault and attributeFormDefault require that the prefix is not
applied to any of the locally declared elements such as shipTo, billTo, name and street, and it is not
applied to any of the attributes (which were all declared locally). The purchaseOrder and comment are
global elements because they are declared in the context of the schema as a whole rather than within the
context of a particular type. For example, the declaration of purchaseOrder appears as a child of the
schema element in po1.xsd, whereas the declaration of shipTo appears as a child of the
complexType element that defines PurchaseOrderType.

When local elements and attributes are not required to be qualified, an instance author may require more or
less knowledge about the details of the schema to create schema valid instance documents. More specifically,
if the author can be sure that only the root element (such as purchaseOrder) is global, then it is a simple
matter to qualify only the root element. Alternatively, the author may know that all the elements are declared
globally, and so all the elements in the instance document can be prefixed, perhaps taking advantage of a
default namespace declaration. (We examine this approach in Section 3.3.) On the other hand, if there is no
uniform pattern of global and local declarations, the author will need detailed knowledge of the schema to
correctly prefix global elements and attributes.

3.2 Qualified Locals

Elements and attributes can be independently required to be qualified, although we start by describing the
qualification of local elements. To specify that all locally declared elements in a schema must be qualified, we
set the value of elementFormDefault to qualified:

Modifications to po1.xsd for Qualified Locals

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:po="http://www.example.com/PO1"
  targetNamespace="http://www.example.com/PO1"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <element name="purchaseOrder" type="po:PurchaseOrderType"/>
  <element name="comment" type="string"/>

  <!−− etc. −−>
</complexType>

<complexType name="PurchaseOrderType">
  <!−− etc. −−>
</complexType>

<schema>

And in this conforming instance document, we qualify all the elements explicitly:

A Purchase Order with Explicitly Qualified Locals

```
<?xml version="1.0"?>
<apo:purchaseOrder xmlns:apo="http://www.example.com/PO1"
  orderDate="1999-10-20">
  <apo:shipTo country="US">
  </apo:shipTo>
</apo:purchaseOrder>
```
Alternatively, we can replace the explicit qualification of every element with implicit qualification provided by a default namespace, as shown here in `po2.xml`:

A Purchase Order with Default Qualified Locals, po2.xml

```xml
<?xml version="1.0"?>
<purchaseOrder xmlns="http://www.example.com/PO1"
  orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <!-- etc. -->
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <!-- etc. -->
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <!-- etc. -->
</purchaseOrder>
```

In `po2.xml`, all the elements in the instance belong to the same namespace, and the namespace statement declares a default namespace that applies to all the elements in the instance. Hence, it is unnecessary to explicitly prefix any of the elements. As another illustration of using qualified elements, the schemas in Section 5 all require qualified elements.

Qualification of attributes is very similar to the qualification of elements. Attributes that must be qualified, either because they are declared globally or because the `attributeFormDefault` attribute is set to qualified, appear prefixed in instance documents. One example of a qualified attribute is the `xsi:nil` attribute that was introduced in Section 2.9. In fact, attributes that are required to be qualified must be explicitly prefixed because the XML–Namespaces specification does not provide a mechanism for defaulting the namespaces of attributes. Attributes that are not required to be qualified appear in instance documents without prefixes, which is the typical case.

The qualification mechanism we have described so far has controlled all local element and attribute declarations within a particular target namespace. It is also possible to control qualification on a declaration by declaration basis using the `form` attribute. For example, to require that the locally declared attribute `publicKey` is qualified in instances, we declare it in the following way:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    attributeFormDefault="qualified">
```

Requiring Qualification of Single Attribute

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    attributeFormDefault="qualified">
```
Notice that the value of the `form` attribute overrides the value of the `attributeFormDefault` attribute for the `publicKey` attribute only. Also, the `form` attribute can be applied to an element declaration in the same manner. An instance document that conforms to the schema is:

Instance with a Qualified Attribute

```xml
<?xml version="1.0"?>
<purchaseOrder xmlns="http://www.example.com/PO1" xmlns:po="http://www.example.com/PO1" orderDate="1999-10-20">
  <!-- etc. -->
  <secure po:publicKey="GpM7">
    <!-- etc. -->
  </secure>
</purchaseOrder>
```

3.3 Global vs. Local Declarations

Another authoring style, applicable when all element names are unique within a namespace, is to create schemas in which all elements are global. This is similar in effect to the use of `<!ELEMENT>` in a DTD. In the example below, we have modified the original `po1.xsd` such that all the elements are declared globally. Notice that we have omitted the `elementFormDefault` and `attributeFormDefault` attributes in this example to emphasize that their values are irrelevant when there are only global element and attribute declarations.

Modified version of `po1.xsd` using only global element declarations

```xml
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:po="http://www.example.com/PO1"
  targetNamespace="http://www.example.com/PO1">
  <element name="purchaseOrder" type="po:PurchaseOrderType"/>

  <element name="shipTo"  type="po:USAddress"/>
  <element name="billTo"  type="po:USAddress"/>
  <element name="comment" type="string"/>

  <element name="name" type="string"/>
  <element name="street" type="string"/>

  <complexType name="PurchaseOrderType">
    <sequence>
    </sequence>
  </complexType>
</schema>
```
This "global" version of po1.xsd will validate the instance document po2.xml which, as we described previously, is also schema valid against the "qualified" version of po1.xsd. In other words, both schema approaches can validate the same, namespace defaulted, document. Thus, in one respect the two schema approaches are similar, although in another important respect the two schema approaches are very different. Specifically, when all elements are declared globally, it is not possible to take advantage of local names. For example, you can only declare one global element called "title". However, you can locally declare one element called "title" that has a string type, and is a subelement of "book". Within the same schema (target namespace) you can declare a second element also called "title" that is an enumeration of the values "Mr Mrs Ms".

### 3.4 Undeclared Target Namespaces

In Section 2 we explained the basics of XML Schema using a schema that did not declare a target namespace and an instance document that did not declare a namespace. So the question naturally arises: What is the target namespace in these examples and how is it referenced?

In the purchase order schema, po.xsd, we did not declare a target namespace for the schema, nor did we declare a prefix (like po: above) associated with the schema's target namespace with which we could refer to types and elements defined and declared within the schema. The consequence of not declaring a target namespace in a schema is that the definitions and declarations from that schema, such as USAddress and purchaseOrder, are referenced without namespace qualification. In other words there is no explicit namespace prefix applied to the references nor is there any implicit namespace applied to the reference by default. So for example, the purchaseOrder element is declared using the type reference PurchaseOrderType. In contrast, all the XML Schema elements and types used in po.xsd are explicitly qualified with the prefix xsd: that is associated with the XML Schema namespace.

In cases where a schema is designed without a target namespace, it is strongly recommended that all XML Schema elements and types are explicitly qualified with a prefix such as xsd: that is associated with the XML Schema namespace (as in po.xsd). The rationale for this recommendation is that if XML Schema elements and types are associated with the XML Schema namespace by default, i.e. without prefixes, then references to XML Schema types may not be distinguishable from references to user-defined types.

Element declarations from a schema with no target namespace validate unqualified elements in the instance document. That is, they validate elements for which no namespace qualification is provided by either an explicit prefix or by default (xmlns:). So, to validate a traditional XML 1.0 document which does not use
namespaces at all, you must provide a schema with no target namespace. Of course, there are many XML 1.0 documents that do not use namespaces, so there will be many schema documents written without target namespaces; you must be sure to give to your processor a schema document that corresponds to the vocabulary you wish to validate.

4. Advanced Concepts II: The International Purchase Order

The purchase order schema described in Chapter 2 was contained in a single document, and most of the schema constructions—such as element declarations and type definitions—were constructed from scratch. In reality, schema authors will want to compose schemas from constructions located in multiple documents, and to create new types based on existing types. In this section, we examine mechanisms that enable such compositions and creations.

4.1 A Schema in Multiple Documents

As schemas become larger, it is often desirable to divide their content among several schema documents for purposes such as ease of maintenance, access control, and readability. For these reasons, we have taken the schema constructs concerning addresses out of po.xsd, and put them in a new file called address.xsd. The modified purchase order schema file is called ipo.xsd:

The International Purchase Order Schema, ipo.xsd

```xml
<schema targetNamespace="http://www.example.com/IPO"
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:ipo="http://www.example.com/IPO">

<documentation xml:lang="en">
    International Purchase order schema for Example.com
    Copyright 2000 Example.com. All rights reserved.
</documentation>
</annotation>

<!−− include address constructs −−>
<include
   schemaLocation="http://www.example.com/schemas/address.xsd"/>

<element name="purchaseOrder" type="ipo:PurchaseOrderType"/>

<element name="comment" type="string"/>

<complexType name="PurchaseOrderType">
   <sequence>
      <element name="shipTo" type="ipo:Address"/>
      <element name="billTo" type="ipo:Address"/>
      <element ref="ipo:comment" minOccurs="0"/>
      <element name="items" type="ipo:Items"/>
   </sequence>
   <attribute name="orderDate" type="date"/>
</complexType>

<complexType name="Items">
   <sequence>
      <element name="item" minOccurs="0" maxOccurs="unbounded">
         <complexType>
            <sequence>
            </sequence>
         </complexType>
      </element>
   </sequence>
</complexType>
```
The file containing the address constructs is:

Addresses for International Purchase Order schema, address.xsd

<schema targetNamespace="http://www.example.com/IPO"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:ipo="http://www.example.com/IPO">

<annotation>
<documentation xml:lang="en">
Addresses for International Purchase order schema
Copyright 2000 Example.com. All rights reserved.
</documentation>
</annotation>

<complexType name="Address">
<sequence>
<element name="name" type="string"/>
<element name="street" type="string"/>
<element name="city" type="string"/>
</sequence>
</complexType>

<complexType name="USAddress">
<complexContent>
<extension base="ipo:Address">
<sequence>
<element name="state" type="ipo:USState"/>
<element name="zip" type="positiveInteger"/>
</sequence>
</extension>
</complexContent>
</complexType>
</schema>
The various purchase order and address constructions are now contained in two schema files, `ipo.xsd` and `address.xsd`. To include these constructions as part of the international purchase order schema, in other words to include them in the international purchase order's namespace, `ipo.xsd` contains the `include` element:

```xml
<include schemaLocation="http://www.example.com/schemas/address.xsd"/>
```

The effect of this `include` element is to bring in the definitions and declarations contained in `address.xsd`, and make them available as part of the international purchase order schema target namespace. The one important caveat to using `include` is that the target namespace of the included components must be the same as the target namespace of the including schema, in this case `http://www.example.com/IPO`. Bringing in definitions and declarations using the `include` mechanism effectively adds these components to the existing target namespace. In Section 4.5, we describe a similar mechanism that enables you to modify certain components when they are brought in.

In our example, we have shown only one including document and one included document. In practice it is possible to include more than one document using multiple `include` elements, and documents can include documents that themselves include other documents. However, nesting documents in this manner is legal only if all the included parts of the schema are declared with the same target namespace.

Instance documents that conform to schema whose definitions span multiple schema documents need only reference the 'topmost' document and the common namespace, and it is the responsibility of the processor to gather together all the definitions specified in the various included documents. In our example above, the instance document `ipo.xml` (see Section 4.3) references only the common target namespace, `http://www.example.com/IPO`, and by implication the one schema file `http://www.example.com/schemas/ipo.xsd`. The processor is responsible for obtaining the schema file `address.xsd`.

In Section 5.4 we describe how schemas can be used to validate content from more than one namespace.
4.2 Deriving Types by Extension

To create our address constructs, we start by creating a complex type called Address in the usual way (see address.xsd). The Address type contains the basic elements of an address: a name, a street and a city. (Such a definition will not work for all countries, but it serves the purpose of our example.) From this starting point we derive two new complex types that contain all the elements of the original type plus additional elements that are specific to addresses in the US and the UK. The technique we use here to derive new (complex) address types by extending an existing type is the same technique we used in Section 2.5.1, except that our base type here is a complex type whereas our base type in the previous section was a simple type.

We define the two new complex types, USAddress and UKAddress, using the complexType element. In addition, we indicate that the content models of the new types are complex, i.e. contain elements, by using thecomplexContent element, and we indicate that we are extending the base type Address by the value of the base attribute on the extension element.

When a complex type is derived by extension, its effective content model is the content model of the base type plus the content model specified in the type derivation. Furthermore, the two content models are treated as two children of a sequential group. In the case of UKAddress, the content model of UKAddress is the content model of Address plus the declarations for a postcode element and an exportCode attribute. This is like defining the UKAddress from scratch as follows:

Example

```xml
<complexType name="UKAddress">
  <sequence>
    <!-- content model of Address -->
    <element name="name" type="string"/>
    <element name="street" type="string"/>
    <element name="city" type="string"/>

    <!-- appended element declaration -->
    <element name="postcode" type="ipo:UKPostcode"/>
  </sequence>

  <!-- appended attribute declaration -->
  <attribute name="exportCode" type="positiveInteger" fixed="1"/>
</complexType>
```

4.3 Using Derived Types in Instance Documents

In our example scenario, purchase orders are generated in response to customer orders which may involve shipping and billing addresses in different countries. The international purchase order, ipo.xml below, illustrates one such case where goods are shipped to the UK and the bill is sent to a US address. Clearly it is better if the schema for international purchase orders does not have to spell out every possible combination of international addresses for billing and shipping, and even more so if we can add new complex types of international address simply by creating new derivations of Address.

XML Schema allows us to define the billTo and shipTo elements as Address types (see ipo.xsd) but to use instances of international addresses in place of instances of Address. In other words, an instance document whose content conforms to the UKAddress type will be valid if that content appears within the document at a location where an Address is expected (assuming the UKAddress content itself is valid). To make this feature of XML Schema work, and to identify exactly which derived type is intended, the
An International Purchase order, ipo.xml

```xml
<?xml version="1.0"?>
<ipo:purchaseOrder
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:ipo="http://www.example.com/IPO"
 orderDate="1999-12-01">
  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>47 Eden Street</street>
    <city>Cambridge</city>
    <postcode>CB1 1JR</postcode>
  </shipTo>
  <billTo xsi:type="ipo:USAddress">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:comment>Want this for the holidays!</ipo:comment>
      <shipDate>1999-12-05</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

In **Section 4.8** we describe how to prevent derived types from being used in this sort of substitution.

### 4.4 Deriving Complex Types by Restriction

In addition to deriving new complex types by extending content models, it is possible to derive new types by restricting the content models of existing types. Restriction of complex types is conceptually the same as restriction of simple types, except that the restriction of complex types involves a type's declarations rather than the acceptable range of a simple type's values. A complex type derived by restriction is very similar to its base type, except that its declarations are more limited than the corresponding declarations in the base type. In fact, the values represented by the new type are a subset of the values represented by the base type (as is the case with restriction of simple types). In other words, an application prepared for the values of the base type would not be surprised by the values of the restricted type.

For example, suppose we want to update our definition of the list of items in an international purchase order so that it must contain **at least one** item on order; the schema shown in **ipo.xsd** allows an items element to appear without any child item elements. To create our new **ConfirmedItems** type, we define the new type in the usual way, indicate that it is derived by restriction from the base type **Items**, and...
provide a new (more restrictive) value for the minimum number of item element occurrences. Notice that types derived by restriction must repeat all the components of the base type definition that are to be included in the derived type:

**Deriving ConfirmedItems by Restriction from Items**

```xml
<complexType name="ConfirmedItems">
  <complexContent>
    <restriction base="ipo:Items">
      <sequence>
        <!−− item element is different than in Items −−>
        <element name="item" minOccurs="1" maxOccurs="unbounded">
          <!−− remainder of definition is same as Items −−>
          <complexType>
            <sequence>
              <element name="productName" type="string"/>
              <element name="quantity">
                <simpleType>
                  <restriction base="positiveInteger">
                    <maxExclusive value="100"/>
                  </restriction>
                </simpleType>
              </element>
              <element name="USPrice" type="decimal"/>
              <element ref="ipo:comment" minOccurs="0"/>
              <element name="shipDate" type="date" minOccurs="0"/>
            </sequence>
            <attribute name="partNum" type="ipo:SKU" use="required"/>
          </complexType>
        </element>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

This change, requiring at least one child element rather than allowing zero or more child elements, narrows the allowable number of child elements from a minimum of 0 to a minimum of 1. Note that all ConfirmedItems type elements will also be acceptable as Item type elements.

To further illustrate restriction, **Table 3** shows several examples of how element and attribute declarations within type definitions may be restricted (the table shows element syntax although the first three examples are equally valid attribute restrictions).

<table>
<thead>
<tr>
<th>Base</th>
<th>Restriction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>default=&quot;1&quot;</td>
<td>setting a default value where none was previously given</td>
<td></td>
</tr>
<tr>
<td>fixed=&quot;100&quot;</td>
<td>setting a fixed value where none was previously given</td>
<td></td>
</tr>
<tr>
<td>type=&quot;string&quot;</td>
<td>specifying a type where none was previously given</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3. Restriction Examples**

4.4 Deriving Complex Types by Restriction
exclusion of an optional component; this may also be accomplished by omitting the component's declaration from the restricted type definition


<table>
<thead>
<tr>
<th>(minOccurs, maxOccurs)</th>
<th>(minOccurs, maxOccurs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 1)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(0, unbounded)</td>
<td>(0, 0) (0, 37)</td>
</tr>
<tr>
<td>(1, 9)</td>
<td>(1, 8) (2, 9) (4, 7) (3, 3)</td>
</tr>
<tr>
<td>(1, unbounded)</td>
<td>(1, 12) (3, unbounded) (6, 6)</td>
</tr>
<tr>
<td>(1, 1)</td>
<td>–</td>
</tr>
</tbody>
</table>

4.5 Redefining Types & Groups

In Section 4.1 we described how to include definitions and declarations obtained from external schema files having the same target namespace. The include mechanism enables you to use externally created schema components "as-is", that is, without any modification. We have just described how to derive new types by extension and by restriction, and the redefine mechanism we describe here enables you to redefine simple and complex types, groups, and attribute groups that are obtained from external schema files. Like the include mechanism, redefine requires the external components to be in the same target namespace as the redefining schema, although external components from schemas that have no namespace can also be redefined. In the latter cases, the redefined components become part of the redefining schema's target namespace.

To illustrate the redefine mechanism, we use it instead of the include mechanism in the International Purchase Order schema, ipo.xsd, and we use it to modify the definition of the complex type Address contained in address.xsd:

Using redefine in the International Purchase Order

```xml
<schema targetNamespace="http://www.example.com/IPO"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ipo="http://www.example.com/IPO">

  <!-- bring in address constructs -->
  <redefine
    schemaLocation="http://www.example.com/schemas/address.xsd">

    <!-- redefinition of Address -->
    <complexType name="Address">
      <complexContent>
        <extension base="ipo:Address">
          <sequence>
            <element name="country" type="string"/>
          </sequence>
        </extension>
      </complexContent>
    </complexType>

  </redefine>

  <!-- etc. -->

</schema>
```

4.5 Redefining Types & Groups
The `redefine` element acts very much like the `include` element as it includes all the declarations and definitions from the `address.xsd` file. The complex type definition of Address uses the familiar extension syntax to add a `country` element to the definition of Address. However, note that the base type is also Address. Outside of the `redefine` element, any such attempt to define a complex type with the same name (and in the same namespace) as the base from which it is being derived would cause an error. But in this case, there is no error, and the extended definition of Address becomes the only definition of Address.

Now that Address has been redefined, the extension applies to all schema components that make use of Address. For example, `address.xsd` contains definitions of international address types that are derived from Address. These derivations reflect the redefined Address type, as shown in the following snippet:

Snippet of `ipo.xml` using Redefined Address

```xml
....
<shipTo exportCode="1" xsi:type="ipo:UKAddress">
  <name>Helen Zoe</name>
  <street>47 Eden Street</street>
  <city>Cambridge</city>
  <!-- country was added to Address which is base type of UKAddress -->
  <country>United Kingdom</country>
  <!-- postcode was added as part of UKAddress -->
  <postcode>CB1 1JR</postcode>
</shipTo>
....
```

Our example has been carefully constructed so that the redefined Address type does not conflict in any way with the types that are derived from the original Address definition. But note that it would be very easy to create a conflict. For example, if the international address type derivations had extended Address by adding a country element, then the redefinition of Address would be adding an element of the same name to the content model of Address. It is illegal to have two elements of the same name (and in the same target namespace) but different types in a content model, and so the attempt to redefine Address would cause an error. In general, `redefine` does not protect you from such errors, and it should be used cautiously.

### 4.6 Substitution Groups

XML Schema provides a mechanism, called substitution groups, that allows elements to be substituted for other elements. More specifically, elements can be assigned to a special group of elements that are said to be substitutable for a particular named element called the head element. (Note that the head element must be declared as a global element.) To illustrate, we declare two elements called `customerComment` and `shipComment` and assign them to a substitution group whose head element is `comment`, and so `customerComment` and `shipComment` can be used anyplace that we are able to use `comment`. Elements in a substitution group must have the same type as the head element, or they can have a type that has been derived from the head element's type. To declare these two new elements, and to make them substitutable for the `comment` element, we use the following syntax:

Declaring Elements Substitutable for `comment`

```xml
<element name="shipComment" type="string"
  substitutionGroup="ipo:comment"/>
<element name="customerComment" type="string"
  substitutionGroup="ipo:comment"/>
```
When these declarations are added to the international purchase order schema, shipComment and customerComment can be substituted for comment in the instance document, for example:

Snippet of ipo.xml with Substituted Elements

....
  <item partNum="833-AA">
    <productName>Lapis necklace</productName>
    <quantity>1</quantity>
    <USPrice>99.95</USPrice>
    <ipo:shipComment>
      Use gold wrap if possible
    </ipo:shipComment>
    <ipo:customerComment>
      Want this for the holidays!
    </ipo:customerComment>
    <shipDate>1999-12-05</shipDate>
  </item>
  ....

Note that when an instance document contains element substitutions whose types are derived from those of their head elements, it is not necessary to identify the derived types using the xsi:type construction that we described in Section 4.3.

The existence of a substitution group does not require any of the elements in that class to be used, nor does it preclude use of the head element. It simply provides a mechanism for allowing elements to be used interchangeably.

4.7 Abstract Elements and Types

XML Schema provides a mechanism to force substitution for a particular element or type. When an element or type is declared to be "abstract", it cannot be used in an instance document. When an element is declared to be abstract, a member of that element's substitution group must appear in the instance document. When an element's corresponding type definition is declared as abstract, all instances of that element must use xsi:type to indicate a derived type that is not abstract.

In the substitution group example we described in Section 4.6, it would be useful to specifically disallow use of the comment element so that instances must make use of the customerComment and shipComment elements. To declare the comment element abstract, we modify its original declaration in the international purchase order schema, ipo.xsd, as follows:

<element name="comment" type="string" abstract="true"/>

With comment declared as abstract, instances of international purchase orders are now only valid if they contain customerComment and shipComment elements.

Declaring an element as abstract requires the use of a substitution group. Declaring a type as abstract simply requires the use of a type derived from it (and identified by the xsi:type attribute) in the instance document. Consider the following schema definition:

Schema for Vehicles
The transport element is not abstract, therefore it can appear in instance documents. However, because its
type definition is abstract, it may never appear in an instance document without an xsi:type attribute that
refers to a derived type. That means the following is not schema−valid:

<transport xmlns="http://cars.example.com/schema"/>

because the transport element's type is abstract. However, the following is schema−valid:

<transport xmlns="http://cars.example.com/schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="Car"/>

because it uses a non−abstract type that is substitutable for Vehicle.

4.8 Controlling the Creation & Use of Derived Types

So far, we have been able to derive new types and use them in instance documents without any restraints. In
reality, schema authors will sometimes want to control derivations of particular types, and the use of derived
types in instances.

XML Schema provides a couple of mechanisms that control the derivation of types. One of these mechanisms
allows the schema author to specify that for a particular complex type, new types may not be derived from it,
either (a) by restriction, (b) by extension, or (c) at all. To illustrate, suppose we want to prevent any derivation
of the Address type by restriction because we intend for it only to be used as the base for extended types
such as USAddress and UKAddress. To prevent any such derivations, we slightly modify the original
definition of Address as follows:

Preventing Derivations by Restriction of Address

<complexType name="Address" final="restriction">
  <sequence>
    <element name="name" type="string"/>
    <element name="street" type="string"/>
    <element name="city" type="string"/>
  </sequence>
</complexType>
The restriction value of the final attribute prevents derivations by restriction. Preventing derivations at all, or by extension, are indicated by the values #all and extension respectively. Moreover, there exists an optional finalDefault attribute on the schema element whose value can be one of the values allowed for the final attribute. The effect of specifying the finalDefault attribute is equivalent to specifying a final attribute on every type definition and element declaration in the schema.

Another type-derivation mechanism controls which facets can be applied in the derivation of a new simple type. When a simple type is defined, the fixed attribute may be applied to any of its facets to prevent a derivation of that type from modifying the value of the fixed facets. For example, we can define a Postcode simple type as:

Preventing Changes to Simple Type Facets

```xml
<simpleType name="Postcode">
    <restriction base="string">
        <length value="7" fixed="true"/>
    </restriction>
</simpleType>
```

Once this simple type has been defined, we can derive a new postal code type in which we apply a facet not fixed in the base definition, for example:

Legal Derivation from Postcode

```xml
<simpleType name="UKPostcode">
    <restriction base="ipo:Postcode">
        <pattern value="[A-Z]{2}\d\s\d[A-Z]{2}"/>
    </restriction>
</simpleType>
```

However, we cannot derive a new postal code in which we re-apply any facet that was fixed in the base definition:

Illegal Derivation from Postcode

```xml
<simpleType name="UKPostcode">
    <restriction base="ipo:Postcode">
        <pattern value="[A-Z]{2}\d\d[A-Z]{2}"/>
        <!— illegal attempt to modify facet fixed in base type —>
        <length value="6" fixed="true"/>
    </restriction>
</simpleType>
```

In addition to the mechanisms that control type derivations, XML Schema provides a mechanism that controls which derivations and substitution groups may be used in instance documents. In Section 4.3, we described how the derived types, USAddress and UKAddress, could be used by the shipTo and billTo elements in instance documents. These derived types can replace the content model provided by the Address type because they are derived from the Address type. However, replacement by derived types can be controlled using the block attribute in a type definition. For example, if we want to block any derivation–by–restriction from being used in place of Address (perhaps for the same reason we defined Address with final="restriction"), we can modify the original definition of Address as follows:
Preventing Derivations by Restriction of Address in the Instance

```xml
<complexType name="Address" block="restriction">
  <sequence>
    <element name="name" type="string"/>
    <element name="street" type="string"/>
    <element name="city" type="string"/>
  </sequence>
</complexType>
```

The `restriction` value on the `_block` attribute prevents derivations—by—restriction from replacing `Address` in an instance. However, it would not prevent `UKAddress` and `USAddress` from replacing `Address` because they were derived by extension. Preventing replacement by derivations at all, or by derivations—by—extension, are indicated by the values `#all` and `extension` respectively. As with `final`, there exists an optional `blockDefault` attribute on the `schema` element whose value can be one of the values allowed for the `block` attribute. The effect of specifying the `blockDefault` attribute is equivalent to specifying a `block` attribute on every type definition and element declaration in the schema.

### 5. Advanced Concepts III: The Quarterly Report

The home—products ordering and billing application can generate ad—hoc reports that summarize how many of which types of products have been billed on a per region basis. An example of such a report, one that covers the fourth quarter of 1999, is shown in `4Q99.xml`.

Notice that in this section we use qualified elements in the schema, and default namespaces where possible in the instances.

**Quarterly Report, 4Q99.xml**

```xml
<purchaseReport
  xmlns="http://www.example.com/Report"
  period="P3M" periodEnding="1999-12-31">
  <regions>
    <zip code="95819">
      <part number="872-AA" quantity="1"/>
      <part number="926-AA" quantity="1"/>
      <part number="833-AA" quantity="1"/>
      <part number="455-BX" quantity="1"/>
    </zip>
    <zip code="63143">
      <part number="455-BX" quantity="4"/>
    </zip>
  </regions>
  <parts>
    <part number="872-AA">Lawnmower</part>
    <part number="926-AA">Baby Monitor</part>
    <part number="833-AA">Lapis Necklace</part>
    <part number="455-BX">Sturdy Shelves</part>
  </parts>
</purchaseReport>
```

The report lists, by number and quantity, the parts billed to various zip codes, and it provides a description of
each part mentioned. In summarizing the billing data, the intention of the report is clear and the data is unambiguous because a number of constraints are in effect. For example, each zip code appears only once (uniqueness constraint). Similarly, the description of every billed part appears only once although parts may be billed to several zip codes (referential constraint), see for example part number 455-BX. In the following sections, we'll see how to specify these constraints using XML Schema.

The Report Schema, report.xsd

```xml
<schema targetNamespace="http://www.example.com/Report"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:r="http://www.example.com/Report"
  xmlns:xipo="http://www.example.com/IPO"
  elementFormDefault="qualified">
  <!−− for SKU −−>
  <import namespace="http://www.example.com/IPO"/>

  <annotation>
    <documentation xml:lang="en">
      Report schema for Example.com
      Copyright 2000 Example.com. All rights reserved.
    </documentation>
  </annotation>

  <element name="purchaseReport">
    <complexType>
      <sequence>
        <element name="regions" type="r:RegionsType">
          <keyref name="dummy2" refer="r:pNumKey">
            <selector xpath="r:zip/r:part"/>
            <field xpath="@number"/>
          </keyref>
        </element>
        <element name="parts" type="r:PartsType"/>
      </sequence>
      <attribute name="period" type="duration"/>
      <attribute name="periodEnding" type="date"/>
    </complexType>
  </element>

  <complexType name="RegionsType">
    <sequence>
      <element name="zip" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="part" maxOccurs="unbounded">
              <complexType>
                <complexContent>
                  XML Schema Part 0: Primer
                </complexContent>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</schema>
```

<restriction base="anyType">
  <attribute name="number" type="xipo:SKU"/>
  <attribute name="quantity" type="positiveInteger"/>
</restriction>
</complexContent>
</complexType>
</element>
</sequence>
</complexType>
</element>
</sequence>
</complexType>
<complexType name="PartsType">
  <sequence>
    <element name="part" maxOccurs="unbounded">
      <complexType>
        <simpleContent>
          <extension base="string">
            <attribute name="number" type="xipo:SKU"/>
          </extension>
        </simpleContent>
      </complexType>
    </element>
  </sequence>
</complexType>
</element>
</sequence>
</complexType>
</complexType>
</schema>

5.1 Specifying Uniqueness

XML Schema enables us to indicate that any attribute or element value must be unique within a certain scope. To indicate that one particular attribute or element value is unique, we use the unique element first to "select" a set of elements, and then to identify the attribute or element "field" relative to each selected element that has to be unique within the scope of the set of selected elements. In the case of our report schema, report.xsd, the selector element's xpath attribute contains an XPath expression, regions/zip, that selects a list of all the zip elements in a report instance. Likewise, the field element's xpath attribute contains a second XPath expression, @code, that specifies that the code attribute values of those elements must be unique. Note that the XPath expressions limit the scope of what must be unique. The report might contain another code attribute, but its value does not have to be unique because it lies outside the scope defined by the XPath expressions. Also note that the XPath expressions you can use in the xpath attribute are limited to a subset of the full XPath language defined in XML Path Language 1.0.

We can also indicate combinations of fields that must be unique. To illustrate, suppose we can relax the constraint that zip codes may only be listed once, although we still want to enforce the constraint that any product is listed only once within a given zip code. We could achieve such a constraint by specifying that the combination of zip code and product number must be unique. From the report document, _4Q99.xml, the combined values of zip code and number would be: {95819 872-AA}, {95819 926-AA}, {95819 833-AA}, {95819 455-BX}, and {63143 455-BX}. Clearly, these combinations do not distinguish between zip code and number combinations derived from single or multiple listings of any particular zip, but the combinations would unambiguously represent a product listed more than once within a single zip. In other words, a schema processor could detect violations of the uniqueness constraint.

To define combinations of values, we simply add field elements to identify all the values involved. So, to
add the part number value to our existing definition, we add a new field element whose xpath attribute value, part/@number, identifies the number attribute of part elements that are children of the zip elements identified by regions/zip:

A Unique Composed Value

```xml
<unique name="dummy1">
  <selector xpath="r:regions/r:zip"/>
  <field xpath="@code"/>
  <field xpath="r:part/@number"/>
</unique>
```

5.2 Defining Keys & their References

In the 1999 quarterly report, the description of every billed part appears only once. We could enforce this constraint using unique, however, we also want to ensure that every part−quantity element listed under a zip code has a corresponding part description. We enforce the constraint using the _key and _keyref elements. The report schema, report.xsd, shows that the _key and _keyref constructions are applied using almost the same syntax as unique. The key element applies to the number attribute value of part elements that are children of the parts element. This declaration of number as a key means that its value must be unique and cannot be set to nil (i.e. is not nillable), and the name that is associated with the key, pNumKey, makes the key referenceable from elsewhere.

To ensure that the part−quantity elements have corresponding part descriptions, we say that the number attribute ( <field>@number</field>) of those elements (<selector>zip/part</selector>) must reference the pNumKey key. This declaration of number as a keyref does not mean that its value must be unique, but it does mean there must exist a pNumKey with the same value.

As you may have figured out by analogy with unique, it is possible to define combinations of _key and _keyref values. Using this mechanism, we could go beyond simply requiring the product numbers to be equal, and define a combination of values that must be equal. Such values may involve combinations of multiple value types (string, integer, date, etc.), provided that the order and type of the field element references is the same in both the _key and _keyref definitions.

5.3 XML Schema Constraints vs. XML 1.0 ID Attributes

XML 1.0 provides a mechanism for ensuring uniqueness using the ID attribute and its associated attributes IDREF and IDREFS. This mechanism is also provided in XML Schema through the ID, IDREF, and IDREFS simple types which can be used for declaring XML 1.0−style attributes. XML Schema also introduces new mechanisms that are more flexible and powerful. For example, XML Schema's mechanisms can be applied to any element and attribute content, regardless of its type. In contrast, ID is a type of attribute and so it cannot be applied to attributes, elements or their content. Furthermore, Schema enables you to specify the scope within which uniqueness applies whereas the scope of an ID is fixed to be the whole document. Finally, Schema enables you to create _keys or a _keyref from combinations of element and attribute content whereas ID has no such facility.

5.4 Importing Types

The report schema, report.xsd, makes use of the simple type xipo:SKU that is defined in another schema, and in another target namespace. Recall that we used include so that the schema in ipo.xsd could make use of definitions and declarations from address.xsd. We cannot use include here
because it can only pull in definitions and declarations from a schema whose target namespace is the same as the including schema's target namespace. Hence, the include element does not identify a namespace (although it does require a schemaLocation). The import mechanism that we describe in this section is an important mechanism that enables schema components from different target namespaces to be used together, and hence enables the schema validation of instance content defined across multiple namespaces.

To import the type SKU and use it in the report schema, we identify the namespace in which SKU is defined, and associate that namespace with a prefix for use in the report schema. Concretely, we use the import element to identify SKU's target namespace, http://www.example.com/IPO, and we associate the namespace with the prefix xipo using a standard namespace declaration. The simple type SKU, defined in the namespace http://www.example.com/IPO, may then be referenced as xipo:SKU in any of the report schema's definitions and declarations.

In our example, we imported one simple type from one external namespace, and used it for declaring attributes. XML Schema in fact permits multiple schema components to be imported, from multiple namespaces, and they can be referred to in both definitions and declarations. For example in report.xsd we could additionally reuse the comment element declared in ipo.xsd by referencing that element in a declaration:

```xml
<element ref="xipo:comment"/>
```

Note however, that we cannot reuse the shipTo element from po.xsd, and the following is not legal because only global schema components can be imported:

```xml
<element ref="xipo:shipTo"/>
```

In ipo.xsd, comment is declared as a global element, in other words it is declared as an element of the schema. In contrast, shipTo is declared locally, in other words it is an element declared inside a complex type definition, specifically the PurchaseOrderType type.

Complex types can also be imported, and they can be used as the base types for deriving new types. Only named complex types can be imported; local, anonymously defined types cannot. Suppose we want to include in our reports the name of an analyst, along with contact information. We can reuse the (globally defined) complex type USAddress from address.xsd, and extend it to define a new type called Analyst by adding the new elements phone and email:

**Defining Analyst by Extending USAddress**

```xml
<complexType name="Analyst">
  <complexContent>
    <extension base="xipo:USAddress">
      <sequence>
        <element name="phone" type="string"/>
        <element name="email" type="string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Using this new type we declare an element called analyst as part of the purchaseReport element declaration (declarations not shown) in the report schema. Then, the following instance document would conform to the modified report schema:
Instance Document Conforming to Report Schema with Analyst Type

```xml
<purchaseReport
xmlns="http://www.example.com/Report"
period="P3M" periodEnding="1999-12-31">
<!-- regions and parts elements omitted -->
<analyst>
  <name>Wendy Uhro</name>
  <street>10 Corporate Towers</street>
  <city>San Jose</city>
  <state>CA</state>
  <zip>95113</zip>
  <phone>408-271-3366</phone>
  <email>uhro@example.com</email>
</analyst>
</purchaseReport>
```

When schema components are imported from multiple namespaces, each namespace must be identified with a separate `import` element. The `import` elements themselves must appear as the first children of the `schema` element. Furthermore, each namespace must be associated with a prefix, using a standard namespace declaration, and that prefix is used to qualify references to any schema components belonging to that namespace. Finally, `import` elements optionally contain a `schemaLocation` attribute to help locate resources associated with the namespaces. We discuss the `schemaLocation` attribute in more detail in a later section.

### 5.4.1 Type Libraries

As XML schemas become more widespread, schema authors will want to create simple and complex types that can be shared and used as building blocks for creating new schemas. XML Schemas already provides types that play this role, in particular, the types described in the Simple Types appendix and in an introductory type library.

Schema authors will undoubtedly want to create their own libraries of types to represent currency, units of measurement, business addresses, and so on. Each library might consist of a schema containing one or more definitions, for example, a schema containing a currency type:

**Example Currency Type in Type Library**

```xml
<schema targetNamespace="http://www.example.com/Currency"
  xmlns:c="http://www.example.com/Currency"
  xmlns="http://www.w3.org/2001/XMLSchema">

  <annotation>
    <documentation xml:lang="en">
      Definition of Currency type based on ISO 4217
    </documentation>
  </annotation>

  <complexType name="Currency">
    <simpleContent>
      <extension base="decimal">
        <attribute name="name">
          <simpleType>
            <restriction base="string">
              <enumeration value="AED">
                XML Schema Part 0: Primer
              </enumeration>
            </restriction>
          </simpleType>
        </attribute>
      </extension>
    </simpleContent>
  </complexType>
</schema>
```
An example of an element appearing in an instance and having this type:

<convertFrom name="AFA">199.37</convertFrom>

Once we have defined the currency type, we can make it available for re-use in other schemas through the import mechanism just described.

### 5.5 Any Element, Any Attribute

In previous sections we have seen several mechanisms for extending the content models of complex types. For example, a mixed content model can contain arbitrary character data in addition to elements, and for example, a content model can contain elements whose types are imported from external namespaces. However, these mechanisms provide very broad and very narrow controls respectively. The purpose of this section is to describe a flexible mechanism that enables content models to be extended by any elements and attributes belonging to specified namespaces.

To illustrate, consider a version of the quarterly report, `4Q99html.xml`, in which we have embedded an HTML representation of the XML parts data. The HTML content appears as the content of the element `htmlExample`, and the default namespace is changed on the outermost HTML element (`<table>`) so that all the HTML elements belong to the HTML namespace, [http://www.w3.org/1999/xhtml](http://www.w3.org/1999/xhtml):

Quarterly Report with HTML, 4Q99html.xml
To permit the appearance of HTML in the instance document we modify the report schema by declaring a new element htmlExample whose content is defined by the _any element. In general, an any element specifies that any well-formed XML is permissible in a type's content model. In the example, we require the XML to belong to the namespace http://www.w3.org/1999/xhtml, in other words, it should be HTML. The example also requires there to be at least one element present from this namespace, as indicated by the values of minOccurs and maxOccurs:

Modification to purchaseReport Declaration to Allow HTML in Instance

```
<element name="purchaseReport">
<complexType>
<sequence>
<element name="regions" type="r:RegionsType"/>
<element name="parts" type="r:PartsType"/>
<element name="htmlExample">
<complexType>
<sequence>
<any namespace="http://www.w3.org/1999/xhtml"
     minOccurs="1" maxOccurs="unbounded"
     processContents="skip"/>
</sequence>
</complexType>
</element>
<attribute name="period" type="duration"/>
<attribute name="periodEnding" type="date"/>
</complexType>
```
The modification permits some well-formed XML belonging to the namespace http://www.w3.org/1999/xhtml to appear inside the htmlExample element. Therefore 4Q99html.xml is permissible because there is one element which (with its children) is well-formed, the element appears inside the appropriate element (htmlExample), and the instance document asserts that the element and its content belongs to the required namespace. However, the HTML may not actually be valid because nothing in 4Q99html.xml by itself can provide that guarantee. If such a guarantee is required, the value of the processContents attribute should be set to strict (the default value). In this case, an XML processor is obliged to obtain the schema associated with the required namespace, and validate the HTML appearing within the htmlExample element.

In another example, we define a text type which is similar to the text type defined in XML Schema's introductory type library (see also Section 5.4.1), and is suitable for internationalized human-readable text. The text type allows an unrestricted mixture of character content and element content from any namespace, for example Ruby annotations, along with an optional xml:lang attribute. The lax value of the processContents attribute instructs an XML processor to validate the element content on a can-do basis: It will validate elements and attributes for which it can obtain schema information, but it will not signal errors for those it cannot obtain any schema information.

**Text Type**

```xml
<xsd:complexType name="text">  
  <xsd:complexContent mixed="true">  
    <xsd:restriction base="xsd:anyType">  
      <xsd:sequence>  
        <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>  
      </xsd:sequence>  
      <xsd:attribute ref="xml:lang"/>  
    </xsd:restriction>  
  </xsd:complexContent>  
</xsd:complexType>
```

Namespaces may be used to permit and forbid element content in various ways depending upon the value of the namespace attribute, as shown in Table 4:

<table>
<thead>
<tr>
<th>Value of Namespace Attribute</th>
<th>Allowable Element Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>##any</td>
<td>Any well-formed XML from any namespace (default)</td>
</tr>
<tr>
<td>##local</td>
<td>Any well-formed XML that is not qualified, i.e. not declared to be in a namespace</td>
</tr>
<tr>
<td>##other</td>
<td>Any well-formed XML that is not from the target namespace of the type being defined</td>
</tr>
<tr>
<td>&quot;<a href="http://www.w3.org/1999/xhtml">http://www.w3.org/1999/xhtml</a> ##targetNamespace&quot;</td>
<td>Any well-formed XML belonging to any namespace in the (whitespace separated) list; ##targetNamespace is shorthand for the target namespace of the type being defined</td>
</tr>
</tbody>
</table>

In addition to the any element which enables element content according to namespaces, there is a corresponding anyAttribute element which enables attributes to appear in elements. For example, we can permit any HTML attribute to appear as part of the htmlExample element by adding anyAttribute to its declaration:
Modification to htmlExample Declaration to Allow HTML Attributes

```
<element name="htmlExample">
  <complexType>
    <sequence>
      <any namespace="http://www.w3.org/1999/xhtml"
        minOccurs="1" maxOccurs="unbounded"
        processContents="skip"/>
    </sequence>
    <anyAttribute namespace="http://www.w3.org/1999/xhtml"/>
  </complexType>
</element>
```

This declaration permits an HTML attribute, say `href`, to appear in the `htmlExample` element. For example:

An HTML attribute in the htmlExample Element

```
....
  <htmlExample xmlns:h="http://www.w3.org/1999/xhtml"
    h:href="http://www.example.com/reports/4Q99.html">
    <!−− HTML markup here −−>
  </htmlExample>
....
```

The `_namespace` attribute in an `_anyAttribute` element can be set to any of the values listed in Table 4 for the `_any` element, and `_anyAttribute` can be specified with a `processContents` attribute. In contrast to an `_any` element, `_anyAttribute` cannot constrain the number of attributes that may appear in an element.

5.6 schemaLocation

XML Schema uses the `schemaLocation` and `xsi:schemaLocation` attributes in three circumstances.

1. In an instance document, the attribute `xsi:schemaLocation` provides hints from the author to a processor regarding the location of schema documents. The author warrants that these schema documents are relevant to checking the validity of the document content, on a namespace by namespace basis. For example, we can indicate the location of the Report schema to a processor of the Quarterly Report:

Using schemaLocation in the Quarterly Report, 4Q99html.xml

```
<purchaseReport
  xmlns="http://www.example.com/Report"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.com/Report
    http://www.example.com/Report.xsd"
  period="P3M" periodEnding="1999-12-31">
  <!−− etc. −−>
</purchaseReport>
```

The `_schemaLocation` attribute contains pairs of values: The first member of each pair is the namespace for which the second member is the hint describing where to find to an appropriate schema document. The
presence of these hints does not require the processor to obtain or use the cited schema documents, and the processor is free to use other schemas obtained by any suitable means, or to use no schema at all.

A schema is not required to have a namespace (see Section 3.4) and so there is a noNamespaceSchemaLocation attribute which is used to provide hints for the locations of schema documents that do not have target namespaces.

2. In a schema, the include element has a required schemaLocation attribute, and it contains a URI reference which must identify a schema document. The effect is to compose a final effective schema by merging the declarations and definitions of the including and the included schemas. For example, in Section 4, the type definitions of Address, USAAddress, UKAddress, USState (along with their attribute and local element declarations) from address.xsd were added to the element declarations of purchaseOrder and comment, and the type definitions of PurchaseOrderType, Items and SKU (along with their attribute and local element declarations) from ipo.xsd to create a single schema.

3. Also in a schema, the import element has optional namespace and schemaLocation attributes. If present, the schemaLocation attribute is understood in a way which parallels the interpretation of xsi:schemaLocation in (1). Specifically, it provides a hint from the author to a processor regarding the location of a schema document that the author warrants supplies the required components for the namespace identified by the namespace attribute. To import components that are not in any target namespace, the import element is used without a namespace attribute (and with or without a schemaLocation attribute). References to components imported in this manner are unqualified.

Note that the schemaLocation is only a hint and some processors and applications will have reasons to not use it. For example, an HTML editor may have a built−in HTML schema.

5.7 Conformance

An instance document may be processed against a schema to verify whether the rules specified in the schema are honored in the instance. Typically, such processing actually does two things, (1) it checks for conformance to the rules, a process called schema validation, and (2) it adds supplementary information that is not immediately present in the instance, such as types and default values, called infoset contributions.

The author of an instance document, such as a particular purchase order, may claim, in the instance itself, that it conforms to the rules in a particular schema. The author does this using the schemaLocation attribute discussed above. But regardless of whether a schemaLocation attribute is present, an application is free to process the document against any schema. For example, a purchasing application may have the policy of always using a certain purchase order schema, regardless of any schemaLocation values.

Conformance checking can be thought of as proceeding in steps, first checking that the root element of the document instance has the right contents, then checking that each subelement conforms to its description in a schema, and so on until the entire document is verified. Processors are required to report what checking has been carried out.

To check an element for conformance, the processor first locates the declaration for the element in a schema, and then checks that the targetNamespace attribute in the schema matches the actual namespace URI of the element. Alternatively, it may determine that the schema does not have a targetNamespace attribute and the instance element is not namespace−qualified.

Supposing the namespaces match, the processor then examines the type of the element, either as given by the declaration in the schema, or by an xsi:type attribute in the instance. If the latter, the instance type must be
an allowed substitution for the type given in the schema; what is allowed is controlled by the _block_ attribute in the element declaration. At this same time, default values and other infoset contributions are applied.

Next the processor checks the immediate attributes and contents of the element, comparing these against the attributes and contents permitted by the element's type. For example, considering a `shipTo` element such as the one in Section 2.1, the processor checks what is permitted for an `Address`, because that is the `shipTo` element's type.

If the element has a simple type, the processor verifies that the element has no attributes or contained elements, and that its character content matches the rules for the simple type. This sometimes involves checking the character sequence against regular expressions or enumerations, and sometimes it involves checking that the character sequence represents a value in a permitted range.

If the element has a complex type, then the processor checks that any required attributes are present and that their values conform to the requirements of their simple types. It also checks that all required subelements are present, and that the sequence of subelements (and any mixed text) matches the content model declared for the complex type. Regarding subelements, schemas can either require exact name matching, permit substitution by an equivalent element or permit substitution by any element allowed by an 'any' particle.

Unless a schema indicates otherwise (as it can for 'any' particles) conformance checking then proceeds one level more deeply by looking at each subelement in turn, repeating the process described above.

### A Acknowledgements

Many people have contributed ideas, material and feedback that has improved this document. In particular, the editor acknowledges contributions from David Beech, Paul Biron, Don Box, Allen Brown, David Cleary, Dan Connolly, Roger Costello, Martin Drst, Martin Gudgin, Dave Hollander, Joe Kesselman, John McCarthy, Andrew Layman, Eve Maler, Ashok Malhotra, Noah Mendelsohn, Michael Sperberg-McQueen, Henry Thompson, Misha Wolf, and Priscilla Walmsley for validating the examples.

### B Simple Types & their Facets

The legal values for each simple type can be constrained through the application of one or more facets. Tables B1.a and B1.b list all of XML Schema's built-in simple types and the facets applicable to each type. The names of the simple types and the facets are linked from the tables to the corresponding descriptions in XML Schema Part 2: Datatypes.

<table>
<thead>
<tr>
<th>Table B1.a. Simple Types &amp; Applicable Facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Types</td>
</tr>
<tr>
<td><strong>Simple Types</strong></td>
</tr>
<tr>
<td>string</td>
</tr>
<tr>
<td>normalizedString</td>
</tr>
<tr>
<td>token</td>
</tr>
<tr>
<td>byte</td>
</tr>
<tr>
<td>unsignedByte</td>
</tr>
<tr>
<td>XML Schema Part 0: Primer</td>
</tr>
<tr>
<td>--------------------------</td>
</tr>
<tr>
<td>base64Binary</td>
</tr>
<tr>
<td>hexBinary</td>
</tr>
<tr>
<td>integer</td>
</tr>
<tr>
<td>positiveInteger</td>
</tr>
<tr>
<td>negativeInteger</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>unsignedInt</td>
</tr>
<tr>
<td>long</td>
</tr>
<tr>
<td>unsignedLong</td>
</tr>
<tr>
<td>short</td>
</tr>
<tr>
<td>unsignedShort</td>
</tr>
<tr>
<td>decimal</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>time</td>
</tr>
<tr>
<td>dateTime</td>
</tr>
<tr>
<td>duration</td>
</tr>
<tr>
<td>date</td>
</tr>
<tr>
<td>gMonth</td>
</tr>
<tr>
<td>gYear</td>
</tr>
<tr>
<td>gYearMonth</td>
</tr>
<tr>
<td>gDay</td>
</tr>
<tr>
<td>gMonthDay</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>QName</td>
</tr>
<tr>
<td>NCName</td>
</tr>
<tr>
<td>anyURI</td>
</tr>
<tr>
<td>language</td>
</tr>
<tr>
<td>ID</td>
</tr>
<tr>
<td>IDREF</td>
</tr>
<tr>
<td>IDREFS</td>
</tr>
</tbody>
</table>
The facets listed in Table B1.b apply only to simple types which are ordered. Not all simple types are ordered and so B1.b does not list all of the simple types.

<table>
<thead>
<tr>
<th>Simple Types</th>
<th>max Inclusive</th>
<th>max Exclusive</th>
<th>min Inclusive</th>
<th>min Exclusive</th>
<th>totalDigits</th>
<th>fractionDigits</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>integer</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>positiveInteger</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>negativeInteger</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>nonNegativeInteger</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>nonPositiveInteger</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>int</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>long</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>short</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>unsignedShort</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>decimal</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>float</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>double</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>time</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>dateTime</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>duration</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>date</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>gMonth</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>gYear</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>

A Acknowledgements 54
C Using Entities

XML 1.0 provides various types of entities which are named fragments of content that can be used in the construction of both DTD's (parameter entities) and instance documents. In Section 2.7, we noted how named groups mimic parameter entities. In this section we show how entities can be declared in instance documents, and how the functional equivalents of entities can be declared in schemas.

Suppose we want to declare and use an entity in an instance document, and that document is also constrained by a schema. For example:

Declaring and referencing an entity in an instance document.

```xml
<?xml version="1.0" ?>
<!DOCTYPE PurchaseOrder [ 
<!ENTITY eacute "é">
]> 
<purchaseOrder xmlns="http://www.example.com/PO1" 
    orderDate="1999−10−20">
    <!-- etc. -->
    <city>Montr&eacute;al</city>
    <!-- etc. -->
</purchaseOrder>
```

Here, we declare an entity called `eacute` as part of an internal (DTD) subset, and we reference this entity in the content of the `city` element. Note that when this instance document is processed, the entity will be dereferenced before schema validation takes place. In other words, a schema processor will determine the validity of the `city` element using `Montréal` as the element's value.

We can achieve a similar but not identical outcome by declaring an element in a schema, and by setting the element's content appropriately:

And this element can be used in an instance document:

Using an element instead of an entity in an instance document.

```xml
<?xml version="1.0" ?>
<purchaseOrder xmlns="http://www.example.com/PO1" 
    xmlns:c="http://www.example.com/characterElements" 
    orderDate="1999−10−20">
    <!-- etc. -->
    <city>Montr<c:eacute/>al</city>
    <!-- etc. -->
</purchaseOrder>
```

In this case, a schema processor will process two elements, a `city` element, and an `eacute` element for the contents of which the processor will supply the single character `é`. Note that the extra element will complicate string matching; the two forms of the name “Montréal” given in the two examples above will not match each other using normal string−comparison techniques.
**D Regular Expressions**

XML Schema's `pattern` facet uses a regular expression language that supports Unicode. It is fully described in XML Schema Part 2. The language is similar to the regular expression language used in the Perl Programming language, although expressions are matched against entire lexical representations rather than user-scoped lexical representations such as line and paragraph. For this reason, the expression language does not contain the metacharacters ^ and $, although ^ is used to express exception, e.g. `[^0-9]x`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Match(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter \d</td>
<td>Chapter 0, Chapter 1, Chapter 2 ....</td>
</tr>
<tr>
<td>Chapter\s\d</td>
<td>Chapter followed by a single whitespace character (space, tab, newline, etc.), followed by a single digit</td>
</tr>
<tr>
<td>Chapter\s\w</td>
<td>Chapter followed by a single whitespace character (space, tab, newline, etc.), followed by a word character (XML 1.0 Letter or Digit)</td>
</tr>
<tr>
<td>Españolía ñola</td>
<td>Española</td>
</tr>
<tr>
<td>\p{Lu}</td>
<td>any uppercase character, the value of \p{} (e.g. &quot;Lu&quot;) is defined by Unicode</td>
</tr>
<tr>
<td>\p{IsGreek}</td>
<td>any Greek character, the 'Is' construction may be applied to any block name (e.g. &quot;Greek&quot;) as defined by Unicode</td>
</tr>
<tr>
<td>\P{IsGreek}</td>
<td>any non-Greek character, the 'Is' construction may be applied to any block name (e.g. &quot;Greek&quot;) as defined by Unicode</td>
</tr>
<tr>
<td>a*x</td>
<td>x, ax, aax, aaax ....</td>
</tr>
<tr>
<td>a?x</td>
<td>ax, x</td>
</tr>
<tr>
<td>a+x</td>
<td>ax, aax, aaax ....</td>
</tr>
<tr>
<td>(a</td>
<td>b)+x</td>
</tr>
<tr>
<td>[abcde]x</td>
<td>ax, bx, cx, dx, ex</td>
</tr>
<tr>
<td>[a−e]x</td>
<td>ax, bx, cx, dx, ex</td>
</tr>
<tr>
<td>[−ae]x</td>
<td>−x, ax, ex</td>
</tr>
<tr>
<td>[ae−]x</td>
<td>ax, ex, −x</td>
</tr>
<tr>
<td>(^0−9)x</td>
<td>any non-digit character followed by the character x</td>
</tr>
<tr>
<td>\Dx</td>
<td>any non-digit character followed by the character x</td>
</tr>
<tr>
<td>.x</td>
<td>any character followed by the character x</td>
</tr>
<tr>
<td>.<em>abc.</em></td>
<td>1x2abc, abc1x2, z3456abchooray ....</td>
</tr>
<tr>
<td>ab{2}x</td>
<td>abbx</td>
</tr>
<tr>
<td>ab{2,4}x</td>
<td>abbx, abbbx, abbbbx</td>
</tr>
<tr>
<td>ab{2,}x</td>
<td>abbx, abbbx, abbbbx ....</td>
</tr>
<tr>
<td>(ab){2}x</td>
<td>ababx</td>
</tr>
</tbody>
</table>
E Index

XML Schema Elements. Each element name is linked to a formal XML description in either the Structures or Datatypes parts of the XML Schema specification. Element names are followed by one or more links to examples (identified by section number) in the Primer.

- all: 2.7
- annotation: 2.6
- any: 5.5
- anyAttribute: 5.5
- appInfo: 2.6
- attribute: 2.2
- attributeGroup: 2.8
- choice: 2.7
- complexContent: 2.5.3
- complexType: 2.2
- documentation: 2.6
- element: 2.2
- enumeration: 2.3
- extension: 2.5.1, 4.2
- field: 5.1
- group: 2.7
- import: 5.4
- include: 4.1
- key: 5.2
- keyref: 5.2
- length: 2.3.1
- list: 2.3.1
- maxInclusive: 2.3
- maxLength: 2.3.1
- minInclusive: 2.3
- minLength: 2.3.1
- pattern: 2.3
- redefine: 4.5
- restriction: 2.3, 4.4
- schema: 2.1
- selector: 5.1
- sequence: 2.7
- simpleContent: 2.5.1
- simpleType: 2.3
- union: 2.3.2
- unique: 5.1

XML Schema Attributes. Each attribute name is followed by one or more pairs of references. Each pair of references consists of a link to an example in the Primer, plus a link to a formal XML description in either the Structures or Datatypes parts of the XML Schema specification.

- abstract: element declaration [Structures], complex type definition [Structures]
- attributeFormDefault: _schema element [Structures]
- base: simple type definition [Datatypes], complex type definition [Structures]
XML Schema's simple types are described in Table 2.
XML Schema Part 0: Primer
Abstract

XML Schema: Structures specifies the XML Schema definition language, which offers facilities for describing the structure and constraining the contents of XML 1.0 documents, including those which exploit the XML Namespace facility. The schema language, which is itself represented in XML 1.0 and uses namespaces, substantially reconstructs and considerably extends the capabilities found in XML 1.0 document type definitions (DTDs). This specification depends on XML Schema Part 2: Datatypes.
The English version of this specification is the only normative version. Information about translations of this document is available at http://www.w3.org/2001/05/xmlschema-translations.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR/.

Table of contents

1 Introduction
1.1 Purpose
1.2 Dependencies on Other Specifications
1.3 Documentation Conventions and Terminology
2 Conceptual Framework
2.1 Overview of XML Schema
2.2 XML Schema Abstract Data Model
2.3 Constraints and Validation Rules
2.4 Conformance
2.5 Names and Symbol Spaces
2.6 Schema–Related Markup in Documents Being Validated
2.7 Representation of Schemas on the World Wide Web
3 Schema Component Details
3.1 Introduction
3.2 Attribute Declarations
3.3 Element Declarations
3.4 Complex Type Definitions
3.5 AttributeUses
3.6 Attribute Group Definitions
3.7 Model Group Definitions
3.8 Model Groups
3.9 Particles
3.10 Wildcards
3.11 Identity–constraint Definitions
3.12 Notation Declarations
3.13 Annotations
3.14 Simple Type Definitions
3.15 Schemas as a Whole
4 Schemas and Namespaces: Access and Composition
4.1 Layer 1: Summary of the Schema–validity Assessment Core
4.2 Layer 2: Schema Documents, Namespaces and Composition
4.3 Layer 3: Schema Document Access and Web–interoperability
5 Schemas and Schema–validity Assessment
5.1 Errors in Schema Construction and Structure
5.2 Assessing Schema–Validity
5.3 Missing Sub–components
5.4 Responsibilities of Schema–aware Processors

Appendices

A Schema for Schemas (normative)
B References (normative)
C Outcome Tabulations (normative)
D Required Information Set Items and Properties (normative)
E Schema Components Diagram (non–normative)
F Glossary (non–normative)
G DTD for Schemas (non–normative)
H Analysis of the Unique Particle Attribution Constraint (non–normative)
I References (non–normative)
J Acknowledgements (non–normative)

1 Introduction

This document sets out the structural part (XML Schema: Structures) of the XML Schema definition language.

Chapter 2 presents a Conceptual Framework (§2) for XML Schemas, including an introduction to the nature of XML Schemas and an introduction to the XML Schema abstract data model, along with other terminology used throughout this document.

Chapter 3, Schema Component Details (§3), specifies the precise semantics of each component of the abstract model, the representation of each component in XML, with reference to a DTD and XML Schema for an XML Schema document type, along with a detailed mapping between the elements and attribute vocabulary of this representation and the components and properties of the abstract model.

Chapter 4 presents Schemas and Namespaces: Access and Composition (§4), including the connection between documents and schemas, the import, inclusion and redefinition of declarations and definitions and the foundations of schema–validity assessment.

Chapter 5 discusses Schemas and Schema–validity Assessment (§5), including the overall approach to schema–validity assessment of documents, and responsibilities of schema–aware processors.

The normative appendices include a Schema for Schemas (normative) (§A) for the XML representation of
schemas and References (normative) (§B).

The non–normative appendices include the DTD for Schemas (non–normative) (§G) and a Glossary (non–normative) (§F).

This document is primarily intended as a language definition reference. As such, although it contains a few examples, it is not primarily designed to serve as a motivating introduction to the design and its features, or as a tutorial for new users. Rather it presents a careful and fully explicit definition of that design, suitable for guiding implementations. For those in search of a step–by–step introduction to the design, the non–normative [XML Schema: Primer] is a much better starting point than this document.

1.1 Purpose

The purpose of XML Schema: Structures is to define the nature of XML schemas and their component parts, provide an inventory of XML markup constructs with which to represent schemas, and define the application of schemas to XML documents.

The purpose of an XML Schema: Structures schema is to define and describe a class of XML documents by using schema components to constrain and document the meaning, usage and relationships of their constituent parts: datatypes, elements and their content and attributes and their values. Schemas may also provide for the specification of additional document information, such as normalization and defaulting of attribute and element values. Schemas have facilities for self–documentation. Thus, XML Schema: Structures can be used to define, describe and catalogue XML vocabularies for classes of XML documents.

Any application that consumes well–formed XML can use the XML Schema: Structures formalism to express syntactic, structural and value constraints applicable to its document instances. The XML Schema: Structures formalism allows a useful level of constraint checking to be described and implemented for a wide spectrum of XML applications. However, the language defined by this specification does not attempt to provide all the facilities that might be needed by any application. Some applications may require constraint capabilities not expressible in this language, and so may need to perform their own additional validations.

1.2 Dependencies on Other Specifications

The definition of XML Schema: Structures depends on the following specifications: [XML−Infoset], [XML−Namespaces], [XPath], and [XML Schemas: Datatypes].

See Required Information Set Items and Properties (normative) (§D) for a tabulation of the information items and properties specified in [XML−Infoset] which this specification requires as a precondition to schema–aware processing.

1.3 Documentation Conventions and Terminology

The section introduces the highlighting and typography as used in this document to present technical material.

Special terms are defined at their point of introduction in the text. For example [Definition:] a term is something used with a special meaning. The definition is labeled as such and the term it defines is displayed in boldface. The end of the definition is not specially marked in the displayed or printed text. Uses of defined terms are links to their definitions, set off with middle dots, for instance ·term·.

Non–normative examples are set off in boxes and accompanied by a brief explanation:
Example

<schema targetNamespace="http://www.example.com/XMLSchema/1.0/mySchema">

And an explanation of the example.

The definition of each kind of schema component consists of a list of its properties and their contents, followed by descriptions of the semantics of the properties:

Schema Component: Example

{example property}
Definition of the property.

References to properties of schema components are links to the relevant definition as exemplified above, set off with curly braces, for instance {example property}.

The correspondence between an element information item which is part of the XML representation of a schema and one or more schema components is presented in a tableau which illustrates the element information item(s) involved. This is followed by a tabulation of the correspondence between properties of the component and properties of the information item. Where context may determine which of several different components may arise, several tabulations, one per context, are given. The property correspondences are normative, as are the illustrations of the XML representation element information items.

In the XML representation, bold−face attribute names (e.g. count below) indicate a required attribute information item, and the rest are optional. Where an attribute information item has an enumerated type definition, the values are shown separated by vertical bars, as for size below; if there is a default value, it is shown following a colon. Where an attribute information item has a built−in simple type definition defined in [XML Schemas: Datatypes], a hyperlink to its definition therein is given.

The allowed content of the information item is shown as a grammar fragment, using the Kleene operators ?, *, and +. Each element name therein is a hyperlink to its own illustration.

NOTE: The illustrations are derived automatically from the Schema for Schemas (normative) (§A). In the case of apparent conflict, the Schema for Schemas (normative) (§A) takes precedence, as it, together with the Schema Representation Constraints, provide the normative statement of the form of XML representations.

XML Representation Summary: example Element Information Item

<example
  count = integer
  size = (large | medium | small) : medium>
Content: (all | any*)
</example>

Example Schema Component

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{example property}</td>
<td>Description of what the property corresponds to, e.g. the value of the size [attribute]</td>
</tr>
</tbody>
</table>
References to elements in the text are links to the relevant illustration as exemplified above, set off with angle brackets, for instance `<example>`.

References to properties of information items as defined in [XML–Infoset] are notated as links to the relevant section thereof, set off with square brackets, for example `[children]`.

Properties which this specification defines for information items are introduced as follows:

PSVI Contributions for example information items

[new property]
  The value the property gets.

References to properties of information items defined in this specification are notated as links to their introduction as exemplified above, set off with square brackets, for example `[new property]`.

The following highlighting is used for non–normative commentary in this document:

NOTE: General comments directed to all readers.

Following [XML 1.0 (Second Edition)], within normative prose in this specification, the words `may` and `must` are defined as follows:

**may**
Conforming documents and XML Schema–aware processors are permitted to but need not behave as described.

**must**
Conforming documents and XML Schema–aware processors are required to behave as described; otherwise they are in error.

Note however that this specification provides a definition of error and of conformant processors' responsibilities with respect to errors (see Schemas and Schema–validity Assessment (§5)) which is considerably more complex than that of [XML 1.0 (Second Edition)].

## 2 Conceptual Framework

This chapter gives an overview of *XML Schema: Structures* at the level of its abstract data model. *Schema Component Details* (§3) provides details on this model, including a normative representation in XML for the components of the model. Readers interested primarily in learning to write schema documents may wish to first read [XML Schema: Primer] for a tutorial introduction, and only then consult the sub–sections of *Schema Component Details* (§3) named XML Representation of ... for the details.

### 2.1 Overview of XML Schema

An XML Schema consists of components such as type definitions and element declarations. These can be used to assess the validity of well–formed element and attribute information items (as defined in [XML–Infoset]), and furthermore may specify augmentations to those items and their descendants. This augmentation makes explicit information which may have been implicit in the original document, such as normalized and/or default values for attributes and elements and the types of element and attribute information items.
Schema–validity assessment has two aspects:

1. determining local schema–validity, that is whether an element or attribute information item satisfies the constraints embodied in the relevant components of an XML Schema;
2. Synthesizing an overall validation outcome for the item, combining local schema–validity with the results of schema–validity assessments of its descendants, if any, and adding appropriate augmentations to the infoset to record this outcome.

Throughout this specification, the word valid and its derivatives are used to refer to clause 1 above, the determination of local schema–validity.

Throughout this specification, the word assessment is used to refer to the overall process of local validation, schema–validity assessment and infoset augmentation.

2.2 XML Schema Abstract Data Model

2.2.1 Type Definition Components 2.2.2 Declaration Components 2.2.3 Model Group Components 2.2.4 Identity–constraint Definition Components 2.2.5 Group Definition Components 2.2.6 Annotation Components

This specification builds on [XML 1.0 (Second Edition)] and [XML−Namespaces]. The concepts and definitions used herein regarding XML are framed at the abstract level of information items as defined in [XML−Infoset]. By definition, this use of the infoset provides a priori guarantees of well−formedness (as defined in [XML 1.0 (Second Edition)]) and namespace conformance (as defined in [XML−Namespaces]) for all candidates for assessment and for all schema documents.

Just as [XML 1.0 (Second Edition)] and [XML−Namespaces] can be described in terms of information items, XML Schemas can be described in terms of an abstract data model. In defining XML Schemas in terms of an abstract data model, this specification rigorously specifies the information which must be available to a conforming XML Schema processor. The abstract model for schemas is conceptual only, and does not mandate any particular implementation or representation of this information. To facilitate interoperability and sharing of schema information, a normative XML interchange format for schemas is provided.

[Definition:] Schema component is the generic term for the building blocks that comprise the abstract data model of the schema. [Definition:] An XML Schema is a set of schema components. There are 13 kinds of component in all, falling into three groups. The primary components, which may (type definitions) or must (element and attribute declarations) have names are as follows:

- Simple type definitions
- Complex type definitions
- Attribute declarations
- Element declarations

The secondary components, which must have names, are as follows:

- Attribute group definitions
- Identity–constraint definitions
- Model group definitions
- Notation declarations

Finally, the "helper" components provide small parts of other components; they are not independent of their context:
During validation, declaration components are associated by (qualified) name to information items being validated.

On the other hand, definition components define internal schema components that can be used in other schema components.

Declarations and definitions may have and be identified by names, which are NCNames as defined by [XML−Namespaces].

Several kinds of component have a target namespace, which is either absent or a namespace name, also as defined by [XML−Namespaces]. The target namespace serves to identify the namespace within which the association between the component and its name exists. In the case of declarations, this in turn determines the namespace name of, for example, the element information items it may validate.

NOTE: At the abstract level, there is no requirement that the components of a schema share a target namespace. Any schema for use in assessment of documents containing names from more than one namespace will of necessity include components with different target namespaces. This contrasts with the situation at the level of the XML representation of components, in which each schema document contributes definitions and declarations to a single target namespace.

Validation, defined in detail in Schema Component Details (§3), is a relation between information items and schema components. For example, an attribute information item may validate with respect to an attribute declaration, a list of element information items may validate with respect to a content model, and so on. The following sections briefly introduce the kinds of components in the schema abstract data model, other major features of the abstract model, and how they contribute to validation.

2.2.1 Type Definition Components

The abstract model provides two kinds of type definition component: simple and complex.

This specification uses the phrase type definition in cases where no distinction need be made between simple and complex types.

Type definitions form a hierarchy with a single root. The subsections below first describe characteristics of that hierarchy, then provide an introduction to simple and complex type definitions themselves.

2.2.1.1 Type Definition Hierarchy

Except for a distinguished ur−type definition, every type definition is, by construction, either a restriction, or an extension, of some other type definition. The graph of these relationships forms a tree known as the Type Definition Hierarchy.
[Definition:] A type definition whose declarations or facets are in a one-to-one relation with those of another specified type definition, with each in turn restricting the possibilities of the one it corresponds to, is said to be a **restriction**. The specific restrictions might include narrowed ranges or reduced alternatives. Members of a type, A, whose definition is a *restriction* of the definition of another type, B, are always members of type B as well.

[Definition:] A complex type definition which allows element or attribute content in addition to that allowed by another specified type definition is said to be an **extension**.

[Definition:] A distinguished ur-type definition is present in each XML Schema, serving as the root of the type definition hierarchy for that schema. The ur-type definition, whose name is *anyType*, has the unique characteristic that it can function as a complex or a simple type definition, according to context. Specifically, *restrictions* of the ur-type definition can themselves be either simple or complex type definitions.

[Definition:] A type definition used as the basis for an *extension* or *restriction* is known as the **base type definition** of that definition.

### 2.2.1.2 Simple Type Definition

A simple type definition is a set of constraints on strings and information about the values they encode, applicable to the *normalized value* of an attribute information item or of an element information item with no element children. Informally, it applies to the values of attributes and the text-only content of elements.

Each simple type definition, whether built-in (that is, defined in [XML Schemas: Datatypes]) or user-defined, is a *restriction* of some particular simple *base type definition*. For the built-in primitive types, this is the simple version of the *ur-type definition*, whose name is *anySimpleType*. This is in turn understood to be a restriction of the *ur-type definition*. Simple types may also be defined whose members are lists of items themselves constrained by some other simple type definition, or whose membership is the union of the memberships of some other simple type definitions. List and union simple type definitions are also understood as restrictions of the simple *ur-type definition*.

For detailed information on simple type definitions, see Simple Type Definitions (§3.14) and [XML Schemas: Datatypes]. The latter also defines an extensive inventory of pre-defined simple types.

### 2.2.1.3 Complex Type Definition

A complex type definition is a set of attribute declarations and a content type, applicable to the [attributes] and [children] of an element information item respectively. The content type may require the [children] to contain neither element nor character information items (that is, to be empty), to be a string which belongs to a particular simple type or to contain a sequence of element information items which conforms to a particular model group, with or without character information items as well.

Each complex type definition is either

- a restriction of a complex *base type definition*.

or

- an *extension* of a simple or complex *base type definition*.

or
• a restriction of the type definition.

A complex type which extends another does so by having additional content model particles at the end of the other definition's content model, or by having additional attribute declarations, or both.

NOTE: This specification allows only appending, and not other kinds of extensions. This decision simplifies application processing required to cast instances from derived to base type. Future versions may allow more kinds of extension, requiring more complex transformations to effect casting.

For detailed information on complex type definitions, see Complex Type Definitions (§3.4).

2.2.2 Declaration Components

There are three kinds of declaration component: element, attribute, and notation. Each is described in a section below. Also included is a discussion of element substitution groups, which is a feature provided in conjunction with element declarations.

2.2.2.1 Element Declaration

An element declaration is an association of a name with a type definition, either simple or complex, an (optional) default value and a (possibly empty) set of identity–constraint definitions. The association is either global or scoped to a containing complex type definition. A top–level element declaration with name ‘A’ is broadly comparable to a pair of DTD declarations as follows, where the associated type definition fills in the ellipses:

```xml
<!ELEMENT A . . .>
<!ATTLIST A . . .>
```

Element declarations contribute to validation as part of model group validation, when their defaults and type components are checked against an element information item with a matching name and namespace, and by triggering identity–constraint definition validation.

For detailed information on element declarations, see Element Declarations (§3.3).

2.2.2.2 Element Substitution Group

In XML 1.0, the name and content of an element must correspond exactly to the element type referenced in the corresponding content model.

[Definition:] Through the new mechanism of element substitution groups, XML Schemas provides a more powerful model supporting substitution of one named element for another. Any top–level element declaration can serve as the defining element, or head, for an element substitution group. Other top–level element declarations, regardless of target namespace, can be designated as members of the substitution group headed by this element. In a suitably enabled content model, a reference to the head validates, not just the head itself, but elements corresponding to any member of the substitution group as well.

All such members must have type definitions which are either the same as the head's type definition or restrictions or extensions of it. Therefore, although the names of elements can vary widely as new namespaces and members of the substitution group are defined, the content of member elements is strictly limited according to the type definition of the substitution group head.
Note that element substitution groups are not represented as separate components. They are specified in the property values for element declarations (see Element Declarations (§3.3)).

2.2.2.3 Attribute Declaration

An attribute declaration is an association between a name and a simple type definition, together with occurrence information and (optionally) a default value. The association is either global, or local to its containing complex type definition. Attribute declarations contribute to validation as part of complex type definition validation, when their occurrence, defaults and type components are checked against an attribute information item with a matching name and namespace.

For detailed information on attribute declarations, see Attribute Declarations (§3.2).

2.2.2.4 Notation Declaration

A notation declaration is an association between a name and an identifier for a notation. For an attribute information item to be valid with respect to a NOTATION simple type definition, its value must have been declared with a notation declaration.

For detailed information on notation declarations, see Notation Declarations (§3.12).

2.2.3 Model Group Components

The model group, particle, and wildcard components contribute to the portion of a complex type definition that controls an element information item's content.

2.2.3.1 Model Group

A model group is a constraint in the form of a grammar fragment that applies to lists of element information items. It consists of a list of particles, i.e. element declarations, wildcards and model groups. There are three varieties of model group:

- Sequence (the element information items match the particles in sequential order);
- Conjunction (the element information items match the particles, in any order);
- Disjunction (the element information items match one of the particles).

For detailed information on model groups, see Model Groups (§3.8).

2.2.3.2 Particle

A particle is a term in the grammar for element content, consisting of either an element declaration, a wildcard or a model group, together with occurrence constraints. Particles contribute to validation as part of complex type definition validation, when they allow anywhere from zero to many element information items or sequences thereof, depending on their contents and occurrence constraints.

[Definition:] A particle can be used in a complex type definition to constrain the validation of the children of an element information item; such a particle is called a content model.

NOTE: XML Schema: Structures -content models- are similar to but more expressive than [XML 1.0 (Second Edition)] content models; unlike [XML 1.0 (Second Edition)], XML Schema: Structures applies -content models- to the -validation- of both mixed and
2.2.3.3 Attribute Use

An attribute use plays a role similar to that of a particle, but for attribute declarations: an attribute declaration within a complex type definition is embedded within an attribute use, which specifies whether the declaration requires or merely allows its attribute, and whether it has a default or fixed value.

2.2.3.4 Wildcard

A wildcard is a special kind of particle which matches element and attribute information items dependent on their namespace name, independently of their local names.

For detailed information on wildcards, see Wildcards (§3.10).

2.2.4 Identity−constraint Definition Components

An identity−constraint definition is an association between a name and one of several varieties of identity−constraint related to uniqueness and reference. All the varieties use [XPath] expressions to pick out sets of information items relative to particular target element information items which are unique, or a key, or a valid reference, within a specified scope. An element information item is only valid with respect to an element declaration with identity−constraint definitions if those definitions are all satisfied for all the descendants of that element information item which they pick out.

For detailed information on identity−constraint definitions, see Identity−constraint Definitions (§3.11).

2.2.5 Group Definition Components

There are two kinds of convenience definitions provided to enable the re−use of pieces of complex type definitions: model group definitions and attribute group definitions.

2.2.5.1 Model Group Definition

A model group definition is an association between a name and a model group, enabling re−use of the same model group in several complex type definitions.

For detailed information on model group definitions, see Model Group Definitions (§3.7).

2.2.5.2 Attribute Group Definition

An attribute group definition is an association between a name and a set of attribute declarations, enabling re−use of the same set in several complex type definitions.

For detailed information on attribute group definitions, see Attribute Group Definitions (§3.6).

2.2.6 Annotation Components

An annotation is information for human and/or mechanical consumers. The interpretation of such information is not defined in this specification.
For detailed information on annotations, see Annotations (§3.13).

### 2.3 Constraints and Validation Rules

The [XML 1.0 (Second Edition)] specification describes two kinds of constraints on XML documents: *well-formedness* and *validity* constraints. Informally, the well-formedness constraints are those imposed by the definition of XML itself (such as the rules for the use of the `<` and `>` characters and the rules for proper nesting of elements), while validity constraints are the further constraints on document structure provided by a particular DTD.

The preceding section focused on validation, that is the constraints on information items which schema components supply. In fact however this specification provides four different kinds of normative statements about schema components, their representations in XML and their contribution to the validation of information items:

**Schema Component Constraint**

[Definition:] Constraints on the schema components themselves, i.e. conditions components must satisfy to be components at all. Located in the sixth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Component Constraints (§C.4).

**Schema Representation Constraint**

[Definition:] Constraints on the representation of schema components in XML beyond those which are expressed in Schema for Schemas (normative) (§A). Located in the third sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Representation Constraints (§C.3).

**Validation Rules**

[Definition:] Contributions to validation associated with schema components. Located in the fourth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Validation Rules (§C.1).

**Schema Information Set Contribution**

[Definition:] Augmentations to post-schema-validation infosets expressed by schema components, which follow as a consequence of validation, and/or assessment. Located in the fifth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Contributions to the post-schema-validation infoset (§C.2).

The last of these, schema information set contributions, are not as new as they might at first seem. XML 1.0 validation augments the XML 1.0 information set in similar ways, for example by providing values for attributes not present in instances, and by implicitly exploiting type information for normalization or access. (As an example of the latter case, consider the effect of NMTOKENS on attribute white space, and the semantics of ID and IDREF.) By including schema information set contributions, this specification makes explicit some features that XML 1.0 left implicit.

### 2.4 Conformance

This specification describes three levels of conformance for schema aware processors. The first is required of all processors. Support for the other two will depend on the application environments for which the processor is intended.

[Definition:] **Minimally conforming** processors must completely and correctly implement the Schema Component Constraints, Validation Rules, and Schema Information Set Contributions contained in this specification.
[Definition:] ·Minimally conforming· processors which accept schemas represented in the form of XML documents as described in Layer 2: Schema Documents, Namespaces and Composition (§4.2) are additionally said to provide **conformance to the XML Representation of Schemas**. Such processors must, when processing schema documents, completely and correctly implement all ·Schema Representation Constraints· in this specification, and must adhere exactly to the specifications in Schema Component Details (§3) for mapping the contents of such documents to ·schema components· for use in ·validation· and ·assessment·.

**NOTE:** By separating the conformance requirements relating to the concrete syntax of XML schema documents, this specification admits processors which use schemas stored in optimized binary representations, dynamically created schemas represented as programming language data structures, or implementations in which particular schemas are compiled into executable code such as C or Java. Such processors can be said to be ·minimally conforming· but not necessarily in ·conformance to the XML Representation of Schemas·.

[Definition:] **Fully conforming** processors are network−enabled processors which are not only both ·minimally conforming· and ·in conformance to the XML Representation of Schemas·, but which additionally must be capable of accessing schema documents from the World Wide Web according to Representation of Schemas on the World Wide Web (§2.7) and How schema definitions are located on the Web (§4.3.2).

**NOTE:** Although this specification provides just these three standard levels of conformance, it is anticipated that other conventions can be established in the future. For example, the World Wide Web Consortium is considering conventions for packaging on the Web a variety of resources relating to individual documents and namespaces. Should such developments lead to new conventions for representing schemas, or for accessing them on the Web, new levels of conformance can be established and named at that time. There is no need to modify or republish this specification to define such additional levels of conformance.

See Schemas and Namespaces: Access and Composition (§4) for a more detailed explanation of the mechanisms supporting these levels of conformance.

### 2.5 Names and Symbol Spaces

As discussed in XML Schema Abstract Data Model (§2.2), most schema components (may) have ·names·. If all such names were assigned from the same "pool", then it would be impossible to have, for example, a simple type definition and an element declaration both with the name "title" in a given ·target namespace·.

Therefore [Definition:] this specification introduces the term **symbol space** to denote a collection of names, each of which is unique with respect to the others. A symbol space is similar to the non−normative concept of namespace partition introduced in [XML−Namespaces]. There is a single distinct symbol space within a given ·target namespace· for each kind of definition and declaration component identified in XML Schema Abstract Data Model (§2.2), except that within a target namespace, simple type definitions and complex type definitions share a symbol space. Within a given symbol space, names are unique, but the same name may appear in more than one symbol space without conflict. For example, the same name can appear in both a type definition and an element declaration, without conflict or necessary relation between the two.

Locally scoped attribute and element declarations are special with regard to symbol spaces. Every complex type definition defines its own local attribute and element declaration symbol spaces, where these symbol spaces are distinct from each other and from any of the other symbol spaces. So, for example, two complex type definitions having the same target namespace can contain a local attribute declaration for the unqualified name "priority", or contain a local element declaration for the name "address", without conflict or necessary relation between the two.
2.6 Schema–Related Markup in Documents Being Validated

2.6.1 xsi:type

The Simple Type Definition (§2.2.1.2) or Complex Type Definition (§2.2.1.3) used in validation of an element is usually determined by reference to the appropriate schema components. An element information item in an instance may, however, explicitly assert its type using the attribute xsi:type. The value of this attribute is a QName; see QName Interpretation (§3.15.3) for the means by which the QName is associated with a type definition.

2.6.2 xsi:nil

XML Schema: Structures introduces a mechanism for signaling that an element should be accepted as valid when it has no content despite a content type which does not require or even necessarily allow empty content. An element may be valid without content if it has the attribute xsi:nil with the value true. An element so labeled must be empty, but can carry attributes if permitted by the corresponding complex type.

2.6.3 xsi:schemaLocation, xsi:noNamespaceSchemaLocation

The xsi:schemaLocation and xsi:noNamespaceSchemaLocation attributes can be used in a document to provide hints as to the physical location of schema documents which may be used for assessment. See How schema definitions are located on the Web (§4.3.2) for details on the use of these attributes.

2.7 Representation of Schemas on the World Wide Web

On the World Wide Web, schemas are conventionally represented as XML documents (preferably of MIME type application/xml or text/xml, but see clause 1.1 of Inclusion Constraints and Semantics (§4.2.1)), conforming to the specifications in Layer 2: Schema Documents, Namespaces and Composition (§4.2). For more information on the representation and use of schema documents on the World Wide Web see Standards for representation of schemas and retrieval of schema documents on the Web (§4.3.1) and How schema definitions are located on the Web (§4.3.2).
3 Schema Component Details

3.1 Introduction

3.1.1 Components and Properties

The following sections provide full details on the composition of all schema components, together with their XML representations and their contributions to `assessment`. Each section is devoted to a single component, with separate subsections for

1. properties: their values and significance
2. XML representation and the mapping to properties
3. constraints on representation
4. validation rules
5. post-schema-validation infoset contributions
6. constraints on the components themselves

The sub-sections immediately below introduce conventions and terminology used throughout the component sections.

3.1.1 Components and Properties

Components are defined in terms of their properties, and each property in turn is defined by giving its range, that is the values it may have. This can be understood as defining a schema as a labeled directed graph, where the root is a schema, every other vertex is a schema component or a literal (string, boolean, number) and every labeled edge is a property. The graph is not acyclic: multiple copies of components with the same name in the same `symbol space` may not exist, so in some cases re-entrant chains of properties must exist. Equality of components for the purposes of this specification is always defined as equality of names (including target namespaces) within symbol spaces.

**NOTE:** A schema and its components as defined in this chapter are an idealization of the information a schema-aware processor requires: implementations are not constrained in how they provide it. In particular, no implications about literal embedding versus indirection follow from the use below of language such as "properties ... having ... components as values".

[Definition:] Throughout this specification, the term `absent` is used as a distinguished property value denoting absence.

Any property not identified as optional is required to be present; optional properties which are not present are taken to have `absent`, as their value. Any property identified as a having a set, subset or list value may have an empty value unless this is explicitly ruled out: this is not the same as `absent`. Any property value identified as a superset or subset of some set may be equal to that set, unless a proper superset or subset is explicitly called for. By 'string' in Part 1 of this specification is meant a sequence of ISO 10646 characters identified as legal XML characters in [XML 1.0 (Second Edition)].
### 3.1.2 XML Representations of Components

The principal purpose of *XML Schema: Structures* is to define a set of schema components that constrain the contents of instances and augment the information sets thereof. Although no external representation of schemas is required for this purpose, such representations will obviously be widely used. To provide for this in an appropriate and interoperable way, this specification provides a normative XML representation for schemas which makes provision for every kind of schema component. [Definition:] A document in this form (i.e. a `<schema>` element information item) is a *schema document*. For the schema document as a whole, and its constituents, the sections below define correspondences between element information items (with declarations in *Schema for Schemas* (normative) (§A) and *DTD for Schemas* (non−normative) (§G)) and schema components. All the element information items in the XML representation of a schema must be in the XML Schema namespace, that is their [namespace name] must be `http://www.w3.org/2001/XMLSchema`. Although a common way of creating the XML Infosets which are or contain -schema documents- will be using an XML parser, this is not required: any mechanism which constructs conformant infosets as defined in [XML−Infoset] is a possible starting point.

Two aspects of the XML representations of components presented in the following sections are constant across them all:

1. All of them allow attributes qualified with namespace names other than the XML Schema namespace itself: these appear as annotations in the corresponding schema component;
2. All of them allow an `<annotation>` as their first child, for human−readable documentation and/or machine−targeted information.

### 3.1.3 The Mapping between XML Representations and Components

For each kind of schema component there is a corresponding normative XML representation. The sections below describe the correspondences between the properties of each kind of schema component on the one hand and the properties of information items in that XML representation on the other, together with constraints on that representation above and beyond those implicit in the *Schema for Schemas* (normative) (§A).

The language used is as if the correspondences were mappings from XML representation to schema component, but the mapping in the other direction, and therefore the correspondence in the abstract, can always be constructed therefrom.

In discussing the mapping from XML representations to schema components below, the value of a component property is often determined by the value of an attribute information item, one of the [attributes] of an element information item. Since schema documents are constrained by the *Schema for Schemas* (normative) (§A), there is always a simple type definition associated with any such attribute information item.

[Definition:] The phrase *actual value* is used to refer to the member of the value space of the simple type definition associated with an attribute information item which corresponds to its -normalized value-. This will often be a string, but may also be an integer, a boolean, a URI reference, etc. This term is also occasionally used with respect to element or attribute information items in a document being validated.

Many properties are identified below as having other schema components or sets of components as values. For the purposes of exposition, the definitions in this section assume that (unless the property is explicitly identified as optional) all such values are in fact present. When schema components are constructed from XML representations involving reference by name to other components, this assumption may be violated if one or more references cannot be resolved. This specification addresses the matter of missing components in a uniform manner, described in Missing Sub−components (§5.3): no mention of handling missing components.
will be found in the individual component descriptions below.

Forward reference to named definitions and declarations is allowed, both within and between schema documents. By the time the component corresponding to an XML representation which contains a forward reference is actually needed for validation, an appropriately−named component may have become available to discharge the reference: see Schemas and Namespaces: Access and Composition (§4) for details.

### 3.1.4 White Space Normalization during Validation

Throughout this specification, the initial value of some attribute information item is the value of the normalized value property of that item. Similarly, the initial value of an element information item is the string composed of, in order, the character code of each character information item in the children of that element information item.

The above definition means that comments and processing instructions, even in the midst of text, are ignored for all validation purposes.

[Definition:] The normalized value of an element or attribute information item is an initial value whose white space, if any, has been normalized according to the value of the whiteSpace facet of the simple type definition used in its validation:

- **preserve**  
  No normalization is done, the value is the normalized value.

- **replace**  
  All occurrences of #x9 (tab), #xA (line feed) and #xD (carriage return) are replaced with #x20 (space).

- **collapse**  
  Subsequent to the replacements specified above under replace, contiguous sequences of #x20s are collapsed to a single #x20, and initial and/or final #x20s are deleted.

There are three alternative validation rules which may supply the necessary background for the above:

- Attribute Locally Valid (§3.2.4) (clause 3)
- Element Locally Valid (Type) (§3.3.4) (clause 3.1.3)
- Element Locally Valid (Complex Type) (§3.4.4) (clause 2.2)

These three levels of normalization correspond to the processing mandated in XML 1.0 for element content, CDATA attribute content and tokenized attributed content, respectively. See Attribute Value Normalization in [XML 1.0 (Second Edition)] for the precedent for replace and collapse for attributes. Extending this processing to element content is necessary to ensure a consistent validation semantics for simple types, regardless of whether they are applied to attributes or elements. Performing it twice in the case of attributes whose normalized value has already been subject to replacement or collapse on the basis of information in a DTD is necessary to ensure consistent treatment of attributes regardless of the extent to which DTD−based information has been made use of during infoset construction.

**NOTE:** Even when DTD−based information has been appealed to, and Attribute Value Normalization has taken place, the above definition of normalized value may mean further normalization takes place, as for instance when character entity references in attribute values result in white space characters other than spaces in their initial values.
3.2 Attribute Declarations

3.2.1 The Attribute Declaration Schema Component

Attribute declarations provide for:

- Local validation of attribute information item values using a simple type definition;
- Specifying default or fixed values for attribute information items.

Example

```xml
<xs:attribute name="age" type="xs:positiveInteger" use="required"/>
```

The XML representation of an attribute declaration.

3.2.1 The Attribute Declaration Schema Component

The attribute declaration schema component has the following properties:

**Schema Component: Attribute Declaration**

- **{name}**: An NCName as defined by [XML-Namespaces].
- **{target namespace}**: Either absent, or a namespace name, as defined in [XML-Namespaces].
- **{type definition}**: A simple type definition.
- **{scope}**: Optional. Either global or a complex type definition.
- **{value constraint}**: Optional. A pair consisting of a value and one of default, fixed.
- **{annotation}**: Optional. An annotation.

The **{name}** property must match the local part of the names of attributes being validated.

The value of the attribute must conform to the supplied **{type definition}**.

A non-absent value of the **{target namespace}** property provides for validation of namespace-qualified attribute information items (which must be explicitly prefixed in the character-level form of XML documents). Absent values of **{target namespace}** validate unqualified (unprefixed) items.

A **{scope}** of global identifies attribute declarations available for use in complex type definitions throughout the schema. Locally scoped declarations are available for use only within the complex type definition identified by the **{scope}** property. This property is absent in the case of declarations within attribute group definitions: their scope will be determined when they are used in the construction of complex type definitions.
The *{value constraint}* reproduces the functions of XML 1.0 default and #FIXED attribute values. *default* specifies that the attribute is to appear unconditionally in the post-schema-validation infoset, with the supplied value used whenever the attribute is not actually present; *fixed* indicates that the attribute value if present must equal the supplied constraint value, and if absent receives the supplied value as for *default*. Note that it is *values* that are supplied and/or checked, not strings.

See Annotations (§3.13) for information on the role of the `{annotation}` property.

**NOTE:** A more complete and formal presentation of the semantics of `{name}`, `{target namespace}` and `{value constraint}` is provided in conjunction with other aspects of complex type ·validation· (see Element Locally Valid (Complex Type) (§3.4.4)).

[XML–Infoset] distinguishes attributes with names such as xmlns or xmlns:xsl from ordinary attributes, identifying them as [namespace attributes]. Accordingly, it is unnecessary and in fact not possible for schemas to contain attribute declarations corresponding to such namespace declarations, see xmlns Not Allowed (§3.2.6). No means is provided in this specification to supply a default value for a namespace declaration.

### 3.2.2 XML Representation of Attribute Declaration Schema Components

The XML representation for an attribute declaration schema component is an `<attribute>` element information item. It specifies a simple type definition for an attribute either by reference or explicitly, and may provide default information. The correspondences between the properties of the information item and properties of the component are as follows:

**XML Representation Summary: attribute Element Information Item**

```xml
<attribute
default = string_
fixed = string_
form = (qualified | unqualified)
id = ID_
name = NCName_
ref = QName_
type = QName_
use = (optional | prohibited | required) : optional

{any attributes with non-schema namespace . . .}>
Content: (annotation?, (simpleType?))
</attribute>
```

If the `<attribute>` element information item has `<schema>` as its parent, the corresponding schema component is as follows:

**Attribute Declaration Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{name}</code></td>
<td>The <em>actual value</em> of the name [attribute]</td>
</tr>
<tr>
<td><code>{target namespace}</code></td>
<td>The <em>actual value</em> of the targetNamespace [attribute] of the parent <code>&lt;schema&gt;</code> element information item, or <em>absent</em> if there is none.</td>
</tr>
<tr>
<td><code>{type definition}</code></td>
<td></td>
</tr>
</tbody>
</table>

3.2 Attribute Declarations
The simple type definition corresponding to the `<simpleType>` element information item in the `children`, if present, otherwise the simple type definition resolved to by the actual value of the type [attribute], if present, otherwise the simple ur-type definition.

{scope} 
  global.

{value constraint} If there is a default or a fixed [attribute], then a pair consisting of the actual value (with respect to the {type definition}) of that [attribute] and either default or fixed, as appropriate, otherwise absent.

{annotation} The annotation corresponding to the `<annotation>` element information item in the `children`, if present, otherwise absent.

otherwise if the `<attribute>` element information item has `<complexType>` or `<attributeGroup>` as an ancestor and the ref [attribute] is absent, it corresponds to an attribute use with properties as follows (unless use='prohibited', in which case the item corresponds to nothing at all):

**Attribute Use Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{required}</td>
<td>true if the use [attribute] is present with actual value: required, otherwise false.</td>
</tr>
<tr>
<td>{attribute declaration}</td>
<td>See the Attribute Declaration mapping immediately below.</td>
</tr>
<tr>
<td>{value constraint}</td>
<td>If there is a default or a fixed [attribute], then a pair consisting of the actual value (with respect to the {type definition} of the {attribute declaration}) of that [attribute] and either default or fixed, as appropriate, otherwise absent.</td>
</tr>
</tbody>
</table>

**Attribute Declaration Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{name}</td>
<td>The actual value of the name [attribute]</td>
</tr>
<tr>
<td>{target namespace}</td>
<td>If form is present and its actual value is qualified, or if form is absent and the actual value of attributeFormDefault on the <code>&lt;schema&gt;</code> ancestor is qualified, then the actual value of the targetNamespace [attribute] of the parent <code>&lt;schema&gt;</code> element information item, or absent if there is none, otherwise absent.</td>
</tr>
<tr>
<td>{type definition}</td>
<td>The simple type definition corresponding to the <code>&lt;simpleType&gt;</code> element information item in the <code>children</code>, if present, otherwise the simple type definition resolved to by the actual value of the type [attribute], if present, otherwise the simple ur-type definition.</td>
</tr>
<tr>
<td>{scope}</td>
<td>If the <code>&lt;attribute&gt;</code> element information item has <code>&lt;complexType&gt;</code> as an ancestor, the complex definition corresponding to that item, otherwise (the <code>&lt;attribute&gt;</code> element information item is within an <code>&lt;attributeGroup&gt;</code> definition), absent.</td>
</tr>
<tr>
<td>{value constraint}</td>
<td>absent.</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotation corresponding to the <code>&lt;annotation&gt;</code> element information item in the <code>children</code>, if present, otherwise absent.</td>
</tr>
</tbody>
</table>

otherwise (the `<attribute>` element information item has `<complexType>` or `<attributeGroup>` as an ancestor)
and the ref \[attribute\] is present), it corresponds to an attribute use with properties as follows (unless use='prohibited', in which case the item corresponds to nothing at all):

**Attribute Use Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{required}</td>
<td>true if the use [attribute] is present with _actual value_ required, otherwise false.</td>
</tr>
<tr>
<td>{attribute declaration}</td>
<td>The (top−level) attribute declaration _resolved_ to by the _actual value_ of the ref [attribute]</td>
</tr>
<tr>
<td>{value constraint}</td>
<td>If there is a default or a fixed [attribute], then a pair consisting of the _actual value_ (with respect to the _type definition_ of the _attribute declaration_) of that [attribute] and either default or fixed, as appropriate, otherwise -absent.</td>
</tr>
</tbody>
</table>

Attribute declarations can appear at the top level of a schema document, or within complex type definitions, either as complete (local) declarations, or by reference to top−level declarations, or within attribute group definitions. For complete declarations, top−level or local, the type attribute is used when the declaration can use a built−in or pre−declared simple type definition. Otherwise an anonymous \(<\text{simpleType}\>) is provided inline.

The default when no simple type definition is referenced or provided is the simple \_ur−type definition\_, which imposes no constraints at all.

Attribute information items \_validated\_ by a top−level declaration must be qualified with the \_target namespace\_ of that declaration (if this is \_absent\_, the item must be unqualified). Control over whether attribute information items \_validated\_ by a local declaration must be similarly qualified or not is provided by the form \[attribute\], whose default is provided by the attributeFormDefault \[attribute\] on the enclosing \(<\text{schema}\>) via its determination of \_target namespace\_.

The names for top−level attribute declarations are in their own \_symbol space\_. The names of locally−scoped attribute declarations reside in symbol spaces local to the type definition which contains them.

### 3.2.3 Constraints on XML Representations of Attribute Declarations

**Schema Representation Constraint: Attribute Declaration Representation OK**

In addition to the conditions imposed on \(<\text{attribute}\>) element information items by the schema for schemas, all of the following must be true:1 default and fixed must not both be present.2 If default and use are both present, use must have the \_actual value\_ optional.3 If the item's parent is not \(<\text{schema}\>) then all of the following must be true:3.1 One of ref or name must be present, but not both.3.2 If ref is present, then all of \(<\text{simpleType}\>\_ form and type must be absent.4 type and \(<\text{simpleType}\>_ must not both be present.5 The corresponding attribute declaration must satisfy the conditions set out in Constraints on Attribute Declaration Schema Components (§3.2.6).

**3.2.4 Attribute Declaration Validation Rules**

**Validation Rule: Attribute Locally Valid**

For an attribute information item to be locally \_valid\_ with respect to an attribute declaration all of the following must be true:1 The declaration must not be \_absent\_ (see Missing Sub−components (§5.3) for how this can fail to be the case).2 Its \_type definition\_ must not be absent.3 The item's \_normalized value\_ must be locally \_valid\_ with respect to that \_type definition\_ as per String Valid (§3.14.4).4 The item's \_actual value\_ must match the value of the \_value constraint\_ if it is present and fixed.

**Validation Rule: Schema−Validity**
Assessment (Attribute)
The schema–validity assessment of an attribute information item depends on its validation, alone.

[Definition:] During validation, associations between element and attribute information items among the [children] and [attributes] on the one hand, and element and attribute declarations on the other, are established as a side-effect. Such declarations are called the context–determined declarations. See clause 3.1 (in Element Locally Valid (Complex Type) (§3.4.4)) for attribute declarations, clause 2 (in Element Sequence Locally Valid (Particle) (§3.9.4)) for element declarations.

For an attribute information item's schema–validity to have been assessed all of the following must be true: 1. A non-absent attribute declaration must be known for it, namely one of the following: 1.1 A declaration which has been established as its context–determined declaration; 1.2 A declaration resolved to by its [local name] and [namespace name] as defined by QName resolution (Instance) (§3.15.4), provided its context–determined declaration is not skip. 2. Its validity with respect to that declaration must have been evaluated as per Attribute Locally Valid (§3.2.4). 3. Both clause 1 and clause 2 of Attribute Locally Valid (§3.2.4) must be satisfied.

[Definition:] For attributes, there is no difference between assessment and strict assessment, so if the above holds, the attribute information item has been strictly assessed.

3.2.5 Attribute Declaration Information Set Contributions

Schema Information Set Contribution: Assessment Outcome (Attribute)
If the schema–validity of an attribute information item has been assessed as per Schema–Validity Assessment (Attribute) (§3.2.4), then in the post–schema–validation infoset it has properties as follows: PSVI Contributions for attribute information items

/validation context/
The nearest ancestor element information item with a [schema information] property.

/validation /
The appropriate case among the following: 1. If it was strictly assessed, then the appropriate case among the following: 1.1 If it was valid; as defined by Attribute Locally Valid (§3.2.4), then valid; 1.2 otherwise invalid. 2. otherwise notKnown.

/validation attempted/
The appropriate case among the following: 1. If it was strictly assessed, then full; 2. otherwise none.

/schema specified /
infoset. See Attribute Default Value (§3.4.5) for the other possible value.

Schema Information Set Contribution: Validation Failure (Attribute)
If the local validity, as defined by Attribute Locally Valid (§3.2.4) above, of an attribute information item has been assessed, in the post–schema–validation infoset the item has a property: PSVI Contributions for attribute information items

/schema error code /
The appropriate case among the following: 1. If the item is not valid, then a list. Applications wishing to provide information as to the reason(s) for the validation failure are encouraged to record one or more error codes (see Outcome Tabulations (normative) (§C)) herein; 2. otherwise absent.

Schema Information Set Contribution: Attribute Declaration
If an attribute information item is valid with respect to an attribute declaration as per Attribute Locally Valid (§3.2.4) then in the post–schema–validation infoset the attribute information item may, at processor option,
have a property: PSVI Contributions for attribute information items

\[\text{attribute declaration}\]
An \text{item isomorphic} to the declaration component itself.

\textbf{Schema Information Set Contribution: Attribute Validated by Type}
If clause 3 of Attribute Locally Valid (§3.2.4) applies with respect to an attribute information item, in the post-schema-validation infoset the attribute information item has a property: PSVI Contributions for attribute information items

\[\text{schema normalized value}\]
The \text{normalized value} of the item as \text{validated}.

Furthermore, the item has one of the following alternative sets of properties:

Either PSVI Contributions for attribute information items

\[\text{type definition}\]
An \text{item isomorphic} to the relevant attribute declaration's \text{type definition} component.

\[\text{member type definition}\]
If and only if that type definition has \text{variety} union, then an \text{item isomorphic} to that member of its \text{member type definitions} which actually validated the attribute item's \text{normalized value}.

or PSVI Contributions for attribute information items

\[\text{type definition type}\]
simple.

\[\text{type definition namespace}\]
The \text{target namespace} of the \text{type definition}.

\[\text{type definition anonymous}\]
\text{true} if the \text{name} of the \text{type definition} is \text{absent}, otherwise \text{false}.

\[\text{type definition name}\]
The \text{name} of the \text{type definition}, if it is not \text{absent}. If it is \text{absent}, schema processors may, but need not, provide a value unique to the definition.

If the \text{type definition} has \text{variety} union, then calling [Definition:] that member of the \text{member type definitions} which actually validated the attribute item's \text{normalized value}, the \textbf{actual member type definition}, there are three additional properties: PSVI Contributions for attribute information items

\[\text{member type definition namespace}\]
The \text{target namespace} of the \text{actual member type definition}.

\[\text{member type definition anonymous}\]
\text{true} if the \text{name} of the \text{actual member type definition} is \text{absent}, otherwise \text{false}.

\[\text{member type definition name}\]
The \text{name} of the \text{actual member type definition}, if it is not \text{absent}. If it is \text{absent}, schema processors may, but need not, provide a value unique to the definition.

The first \text{item isomorphic} alternative above is provided for applications such as query processors which need access to the full range of details about an item's \text{assessment}, for example the type hierarchy; the second, for lighter-weight processors for whom representing the significant parts of the type hierarchy as information items might be a significant burden.

3.2 Attribute Declarations
Also, if the declaration has a \{value constraint\}, the item has a property: PSVI Contributions for attribute information items

\[schema
default\]

The canonical lexical representation of the declaration's \{value constraint\} value.

If the attribute information item was not \strictly assessed, then instead of the values specified above,1 The item's \{schema normalized value\} property has the \initial value\ of the item as its value;2 The \{type definition\} and \{member type definition\} properties, or their alternatives, are based on the \simple ur\−type definition.3

3.2.6 Constraints on Attribute Declaration Schema Components

All attribute declarations (see Attribute Declarations (§3.2)) must satisfy the following constraints.

**Schema Component Constraint: Attribute Declaration Properties Correct**

All of the following must be true:1 The values of the properties of an attribute declaration must be as described in the property tableau in The Attribute Declaration Schema Component (§3.2.1), modulo the impact of Missing Sub−components (§5.3).2 if there is a \{value constraint\}, the canonical lexical representation of its value must be \valid\; with respect to the \{type definition\} as defined in String Valid (§3.14.4).3 If the \{type definition\} is or is derived from ID then there must not be a \{value constraint\}.

**Schema Component Constraint: xmlns Not Allowed**

The \{name\} of an attribute declaration must not match xmlns.

**NOTE:** The \{name\} of an attribute is an \NCName, which implicitly prohibits attribute declarations of the form xmlns:*

**Schema Component Constraint: xsi: Not Allowed**

The \{target namespace\} of an attribute declaration, whether local or top−level, must not match http://www.w3.org/2001/XMLSchema−instance (unless it is one of the four built−in declarations given in the next section).

**NOTE:** This reinforces the special status of these attributes, so that they not only need not be declared to be allowed in instances, but must not be declared. It also removes any temptation to experiment with supplying global or fixed values for e.g. xsi:type or xsi:nil, which would be seriously misleading, as they would have no effect.

3.2.7 Built−in Attribute Declarations

There are four attribute declarations present in every schema by definition:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{name}</td>
<td>type</td>
</tr>
<tr>
<td>{target namespace}</td>
<td><a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a></td>
</tr>
<tr>
<td>{type definition}</td>
<td>The built−in QName simple type definition</td>
</tr>
<tr>
<td>{scope}</td>
<td>global</td>
</tr>
</tbody>
</table>
### Attribute Declaration for the 'nil' attribute

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{name}</td>
<td>nil</td>
</tr>
<tr>
<td>{target namespace}</td>
<td><a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a></td>
</tr>
<tr>
<td>{type definition}</td>
<td>The built-in <code>boolean</code> simple type definition</td>
</tr>
<tr>
<td>{scope}</td>
<td><code>global</code></td>
</tr>
<tr>
<td>{value constraint}</td>
<td>:absent.</td>
</tr>
<tr>
<td>{annotation}</td>
<td>:absent.</td>
</tr>
</tbody>
</table>

### Attribute Declaration for the 'schemaLocation' attribute

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{name}</td>
<td><code>schemaLocation</code></td>
</tr>
<tr>
<td>{target namespace}</td>
<td><a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a></td>
</tr>
</tbody>
</table>

An anonymous simple type definition, as follows:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{name}</td>
<td>:absent.</td>
</tr>
<tr>
<td>{target namespace}</td>
<td><a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a></td>
</tr>
<tr>
<td>{type definition}</td>
<td>The built-in <code>simple ur-type definition</code>.</td>
</tr>
<tr>
<td>{base type definition}</td>
<td></td>
</tr>
<tr>
<td>{facets}</td>
<td>:absent.</td>
</tr>
<tr>
<td>{variety}</td>
<td><code>list</code></td>
</tr>
<tr>
<td>{item type definition}</td>
<td>The built-in <code>anyURI</code> simple type definition</td>
</tr>
<tr>
<td>{annotation}</td>
<td>:absent.</td>
</tr>
<tr>
<td>{scope}</td>
<td><code>global</code></td>
</tr>
<tr>
<td>{value constraint}</td>
<td>:absent.</td>
</tr>
<tr>
<td>{annotation}</td>
<td>:absent.</td>
</tr>
</tbody>
</table>

### Attribute Declaration for the 'noNamespaceSchemaLocation' attribute
### 3.3 Element Declarations

#### 3.3.1 The Element Declaration Schema Component

The element declaration schema component has the following properties:

- **{name}**
  - An NCName as defined by [XML−Namespaces].

- **{target namespace}**
  - Either **absent** or a namespace name, as defined in [XML−Namespaces].

#### 3.3.2 XML Representation of Element Declaration

Example

```xml
<xs:element name="PurchaseOrder" type="PurchaseOrderType"/>
<xs:element name="gift">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="birthday" type="xs:date"/>
      <xs:element ref="PurchaseOrder"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

XML representations of several different types of element declaration

### 3.3.3 Constraints on XML Representations of Element Declarations

Element declarations provide for:

- Local **validation** of element information item values using a type definition;
- Specifying default or fixed values for an element information items;
- Establishing uniquenesses and reference constraint relationships among the values of related elements and attributes;
- Controlling the substitutability of elements through the mechanism of **element substitution groups**.

#### Example

```xml
<xs:element name="PurchaseOrder" type="PurchaseOrderType"/>
<xs:element name="gift">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="birthday" type="xs:date"/>
      <xs:element ref="PurchaseOrder"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

### 3.3.4 Element Declaration Validation Rules

### 3.3.5 Element Declaration Information Set Contributions

### 3.3.6 Constraints on Element Declaration Schema Components
Either a simple type definition or a complex type definition.

Optional. Either global or a complex type definition.

Optional. A pair consisting of a value and one of default, fixed.

A boolean.

A set of constraint definitions.

Optional. A top-level element definition.

A subset of \{extension, restriction\}.

A subset of \{substitution, extension, restriction\}.

A boolean.

Optional. An annotation.

The \{name\} property must match the local part of the names of element information items being validated.

A \{scope\} of global identifies element declarations available for use in content models throughout the schema. Locally scoped declarations are available for use only within the complex type identified by the \{scope\} property. This property is absent in the case of declarations within named model groups: their scope is determined when they are used in the construction of complex type definitions.

A non-absent value of the \{target namespace\} property provides for validation of namespace-qualified element information items. Absent values of \{target namespace\} validate unqualified items.

An element information item is valid if it satisfies the \{type definition\}. For such an item, schema information set contributions appropriate to the \{type definition\} are added to the corresponding element information item in the post-schema-validation infoset.

If \{nillable\} is true, then an element may also be valid: if it carries the namespace qualified attribute with \{local name\} nil from namespace http://www.w3.org/2001/XMLSchema-instance and value true (see xsi:nil (§2.6.2)) even if it has no text or element content despite a \{content type\} which would otherwise require content. Formal details of element validation are described in Element Locally Valid (Element) (§3.3.4).

\{value constraint\} establishes a default or fixed value for an element. If default is specified, and if the element being validated is empty, then the canonical form of the supplied constraint value becomes the [schema normalized value] of the validated element in the post-schema-validation infoset. If fixed is specified, then the element's content must either be empty, in which case fixed behaves as default, or its value must match the supplied constraint value.

NOTE: The provision of defaults for elements goes beyond what is possible in XML 1.0 DTDs, and does not exactly correspond to defaults for attributes. In particular, an element with a non-empty \{value constraint\} whose simple type definition includes the empty string in its lexical space will nonetheless never receive that value, because the \{value constraint\}
Element declarations are members of the substitution group, if any, identified by `{substitution group affiliation}`. Membership is transitive but not symmetric; an element declaration is a member of any group of which its `{substitution group affiliation}` is a member.

An empty `{substitution group exclusions}` allows a declaration to be nominated as the `{substitution group affiliation}` of other element declarations having the same `{type definition}` or types derived therefrom. The explicit values of `{substitution group exclusions}` rule out element declarations having types which are extensions or restrictions respectively of `{type definition}`. If both values are specified, then the declaration may not be nominated as the `{substitution group affiliation}` of any other declaration.

The supplied values for `{disallowed substitutions}` determine whether an element declaration appearing in a ·content model· will be prevented from additionally ·validating· elements (a) with an xsi:type (§2.6.1) that identifies an extension or restriction of the type of the declared element, and/or (b) from validating elements which are in the substitution group headed by the declared element. If `{disallowed substitutions}` is empty, then all derived types and substitution group members are allowed.

Element declarations for which `{abstract}` is true can appear in content models only when substitution is allowed; such declarations may not themselves ever be used to validate element content.

See Annotations (§3.13) for information on the role of the `{annotation}` property.

### 3.3.2 XML Representation of Element Declaration Schema Components

The XML representation for an element declaration schema component is an `<element>` element information item. It specifies a type definition for an element either by reference or explicitly, and may provide occurrence and default information. The correspondences between the properties of the information item and properties of the component(s) it corresponds to are as follows:

XML Representation Summary: `<element>` Element Information Item

```xml
<element
    abstract = boolean : false
    block = (#all | List of (extension | restriction | substitution))
    default = string
    final = (#all | List of (extension | restriction))
    fixed = string
    form = (qualified | unqualified)
    id = ID
    maxOccurs = (nonNegativeInteger | unbounded) : 1
    minOccurs = nonNegativeInteger : 1
    name = NCName
    nillable = boolean : false
    ref = QName
    substitutionGroup = QName
    type = QName
    {any attributes with non-schema namespace . . .}>
```
If the `<element>` element information item has `<schema>` as its parent, the corresponding schema component is as follows:

**Element Declaration Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{name}</code></td>
<td>The ·actual value· of the name [attribute].</td>
</tr>
<tr>
<td><code>{target namespace}</code></td>
<td>The ·actual value· of the targetNamespace [attribute] of the parent <code>&lt;schema&gt;</code> element information item, or ·absent· if there is none.</td>
</tr>
<tr>
<td><code>{scope}</code></td>
<td>global.</td>
</tr>
<tr>
<td><code>{type definition}</code></td>
<td>The type definition corresponding to the <code>&lt;simpleType&gt;</code> or <code>&lt;complexType&gt;</code> element information item in the [children], if either is present, otherwise the type definition ·resolved· to by the ·actual value· of the type [attribute], otherwise the <code>{type definition}</code> of the element declaration ·resolved· to by the ·actual value· of the substitutionGroup [attribute], if present, otherwise the ·ur−type definition·.</td>
</tr>
<tr>
<td><code>{nillable}</code></td>
<td>The ·actual value· of the nillable [attribute], if present, otherwise false.</td>
</tr>
<tr>
<td><code>{value constraint}</code></td>
<td>If there is a default or a fixed [attribute], then a pair consisting of the ·actual value· (with respect to the <code>{type definition}</code>, if it is a simple type definition, or the <code>{type definition}</code>'s <code>{content type}</code>, if that is a simple type definition, or else with respect to the built−in string simple type definition) of that [attribute] and either default or fixed, as appropriate, otherwise ·absent·.</td>
</tr>
<tr>
<td><code>{identity−constraint definitions}</code></td>
<td>A set consisting of the identity−constraint−definitions corresponding to all the <code>&lt;key&gt;</code>, <code>&lt;unique&gt;</code> and <code>&lt;keyref&gt;</code> element information items in the [children], if any, otherwise the empty set.</td>
</tr>
<tr>
<td><code>{substitution group affiliation}</code></td>
<td>The element declaration ·resolved· to by the ·actual value· of the substitutionGroup [attribute], if present, otherwise ·absent·.</td>
</tr>
<tr>
<td><code>{disallowed substitutions}</code></td>
<td>A set depending on the ·actual value· of the block [attribute], if present, otherwise on the ·actual value· of the blockDefault [attribute] of the ancestor <code>&lt;schema&gt;</code> element information item, if present, otherwise on the empty string. Call this the EBV (for effective block value). Then the value of this property is the appropriate case among the following:1 If the EBV is the empty string, then the empty set;2 If the EBV is #all, then {extension, restriction, substitution};3 otherwise a set with members drawn from the set above, each being present or absent depending on whether the ·actual value· (which is a list) contains an equivalently named item.</td>
</tr>
</tbody>
</table>
NOTE: Although the blockDefault [attribute] of <schema> may include values other than extension, restriction or substitution, those values are ignored in the determination of {disallowed substitutions} for element declarations (they are used elsewhere).

{substitution group exclusions} As for {disallowed substitutions} above, but using the final and finalDefault [attributes] in place of the block and blockDefault [attributes] and with the relevant set being {extension, restriction}.

{abstract} The -actual value- of the abstract [attribute], if present, otherwise false.

{annotation} The annotation corresponding to the <annotation> element information item in the [children], if present, otherwise _absent_.

otherwise if the <element> element information item has <complexType> or <group> as an ancestor and the ref [attribute] is absent, the corresponding schema components are as follows (unless minOccurs=maxOccurs=0, in which case the item corresponds to no component at all):

**Particle Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{min occurs}</td>
<td>The ·actual value· of the minOccurs [attribute], if present, otherwise 1.</td>
</tr>
<tr>
<td>{max occurs}</td>
<td>unbounded, if the maxOccurs [attribute] equals unbounded, otherwise the ·actual value· of the maxOccurs [attribute], if present, otherwise 1.</td>
</tr>
<tr>
<td>{term}</td>
<td>A (local) element declaration as given below.</td>
</tr>
</tbody>
</table>

An element declaration as in the first case above, with the exception of its {target namespace} and {scope} properties, which are as below:

**Element Declaration Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{target namespace}</td>
<td>If form is present and its ·actual value· is qualified, or if form is absent and the ·actual value· of elementFormDefault on the &lt;schema&gt; ancestor is qualified, then the ·actual value· of the targetNamespace [attribute] of the parent &lt;schema&gt; element information item, or ·absent· if there is none, otherwise ·absent·.</td>
</tr>
<tr>
<td>{scope}</td>
<td>If the &lt;element&gt; element information item has &lt;complexType&gt; as an ancestor, the complex definition corresponding to that item, otherwise (the &lt;element&gt; element information item is within a named &lt;group&gt; definition), ·absent·.</td>
</tr>
</tbody>
</table>

otherwise (the <element> element information item has <complexType> or <group> as an ancestor and the ref [attribute] is present), the corresponding schema component is as follows (unless minOccurs=maxOccurs=0, in which case the item corresponds to no component at all):

**Particle Schema Component**
Property | Representation
--- | ---
(min occurs) | The ·actual value· of the minOccurs [attribute], if present, otherwise 1.
(max occurs) | unbounded, if the maxOccurs [attribute] equals unbounded, otherwise the ·actual value· of the maxOccurs [attribute], if present, otherwise 1.
(term) | The (top−level) element declaration ·resolved· to by the ·actual value· of the ref [attribute].

<element> corresponds to an element declaration, and allows the type definition of that declaration to be specified either by reference or by explicit inclusion.

<element>s within <schema> produce global element declarations; <element>s within <group> or <complexType> produce either particles which contain global element declarations (if there's a ref attribute) or local declarations (otherwise). For complete declarations, top−level or local, the type attribute is used when the declaration can use a built−in or pre−declared type definition. Otherwise an anonymous <simpleType> or <complexType> is provided inline.

Element information items ·validated· by a top−level declaration must be qualified with the {target namespace} of that declaration (if this is ·absent·, the item must be unqualified). Control over whether element information items ·validated· by a local declaration must be similarly qualified or not is provided by the form [attribute], whose default is provided by the elementFormDefault [attribute] on the enclosing <schema>, via its determination of {target namespace}.

As noted above the names for top−level element declarations are in a separate ·symbol space· from the symbol spaces for the names of type definitions, so there can (but need not be) a simple or complex type definition with the same name as a top−level element. As with attribute names, the names of locally−scoped element declarations with no {target namespace} reside in symbol spaces local to the type definition which contains them.

Note that the above allows for two levels of defaulting for unspecified type definitions. An <element> with no referenced or included type definition will correspond to an element declaration which has the same type definition as the head of its substitution group if it identifies one, otherwise the ·ur−type definition·. This has the important consequence that the minimum valid element declaration, that is, one with only a name attribute and no contents, is also the most general, validating any combination of text and element content and allowing any attributes.

See below at XML Representation of Identity−constraint Definition Schema Components (§3.11.2) for <key>, <unique> and <keyref>.

Example

```xml
<xs:element name="unconstrained"/>
<xs:element name="emptyElt">
  <xs:complexType>
    <xs:attribute .../>. . .</xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="contextOne">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="myLocalElement" type="myFirstType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```
The first example above declares an element whose type, by default, is the *ur-type definition*. The second uses an embedded anonymous complex type definition.

The last two examples illustrate the use of local element declarations. Instances of `myLocalElement` within `contextOne` will be constrained by `myFirstType`, while those within `contextTwo` will be constrained by `mySecondType`.

**NOTE:** The possibility that differing attribute declarations and/or content models would apply to elements with the same name in different contexts is an extension beyond the expressive power of a DTD in XML 1.0.

**Example**

```xml
<xs:complexType name="facet">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:attribute name="value" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="facet" type="xs:facet" abstract="true"/>

<xs:element name="encoding" substitutionGroup="xs:facet">
  <xs:complexType>
    <xs:complexContent>
      <xs:restriction base="xs:facet">
        <xs:sequence>
          <xs:element ref="annotation" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="value" type="xs:encodings"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="period" substitutionGroup="xs:facet">
  <xs:complexType>
    <xs:complexContent>
      <xs:restriction base="xs:facet">
        <xs:sequence>
          <xs:element ref="annotation" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="value" type="xs:duration"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```
An example from a previous version of the schema for datatypes. The facet type is defined and the facet element is declared to use it. The facet element is abstract — it's only defined to stand as the head for a substitution group. Two further elements are declared, each a member of the facet substitution group. Finally a type is defined which refers to facet, thereby allowing either period or encoding (or any other member of the group).

3.3.3 Constraints on XML Representations of Element Declarations

Schema Representation Constraint: Element Declaration Representation OK

In addition to the conditions imposed on <element> element information items by the schema for schemas: all of the following must be true:

1. default and fixed must not both be present.
2. If the item's parent is not <schema>, then all of the following must be true:
   2.1 One of ref or name must be present, but not both.
   2.2 If ref is present, then all of <complexType>, <simpleType>, <key>, <keyref>, <unique>, nillable, default, fixed, form, block and type must be absent, i.e. only minOccurs, maxOccurs, id are allowed in addition to ref, along with <annotation>.
3. type and either <simpleType> or <complexType> are mutually exclusive.
4. The corresponding particle and/or element declarations must satisfy the conditions set out in Constraints on Element Declaration Schema Components (§3.3.6) and Constraints on Particle Schema Components (§3.9.6).

3.3.4 Element Declaration Validation Rules

Validation Rule: Element Locally Valid (Element)

For an element information item to be locally valid with respect to an element declaration all of the following must be true:

1. The declaration must not be absent. 2 Its {abstract} must be false.
2. If {nillable} is false, then there must be no attribute information item among the element information item's {attributes} whose {namespace name} is identical to http://www.w3.org/2001/XMLSchema-instance and whose {local name} is nil.
3.2 If {nillable} is true and there is such an attribute information item and its {actual value} is true, then all of the following must be true:
   3.2.1 The element information item must have no character or element information item {children}.
   3.2.2 There must be no fixed {value constraint}.
4. If there is an attribute information item among the element information item's {attributes} whose {namespace name} is identical to http://www.w3.org/2001/XMLSchema-instance and whose {local name} is type, then all of the following must be true:
   4.1 The {normalized value} of that attribute information item must be valid, with respect to the built-in QName simple type, as defined by String Valid (§3.14.4);
   4.2 The {local name} and {namespace name} (as defined in QName Interpretation (§3.15.3)), of the {actual value} of that attribute information item must resolve to a type definition, as defined in QName resolution (Instance) (§3.15.4) — [Definition:] call this type definition the local type definition.
   4.3 The local type definition must be validly derived from the {type definition}, given the union of the {disallowed substitutions} and the {type definition}'s {prohibited substitutions}, as defined in Type Derivation OK (Complex) (§3.4.6) (if it is a complex type definition), or given {disallowed substitutions} as defined in Type Derivation OK (Simple)
The phrase actual type definition occurs below. If the above three clauses are satisfied, this should be understood as referring to the local type definition, otherwise to the type definition. If the declaration has a value constraint, the item has neither element nor character [children] and clause 3.2 has not applied, then all of the following must be true: If the actual type definition is a local type definition, then the canonical lexical representation of the value constraint value must be a valid default for the actual type definition, as defined in Element Default Valid (Immediate) (§3.3.6). The element information item with the canonical lexical representation of the value constraint value used as its normalized value must be valid with respect to the actual type definition, as defined by Element Locally Valid (Type) (§3.3.4). If there is a fixed value constraint and clause 3.2 has not applied, all of the following must be true: If the content type of the actual type definition is mixed, then the initial value of the item must match the canonical lexical representation of the value constraint value. If the content type of the actual type definition is a simple type definition, then the actual value of the item must match the canonical lexical representation of the value constraint value. The element information item must be valid with respect to each of the identity-constraint definitions as per Identity-constraint Satisfied (§3.11.4). If the element information item is the validation root, it must be valid per Validation Root Valid (ID/IDREF) (§3.3.4). Validation Rule: Element Locally Valid (Type)

For an element information item to be locally valid with respect to a type definition all of the following must be true: 1. The type definition must not be absent; 2. Its abstract must be false. The appropriate case among the following must be true: If the type definition is a simple type definition, then all of the following must be true: 1. The element information item's attributes must be empty, excepting those whose namespace name is identical to http://www.w3.org/2001/XMLSchema-instance and whose local name is one of type, nil, schemaLocation or noNamespaceSchemaLocation. 2. If clause 3.2 of Element Locally Valid (Element) (§3.3.4) did not apply, then the normalized value must be valid with respect to the type definition. 3. If the type definition is a complex type definition, then the element information item must be valid with respect to the type definition as per Element Locally Valid (Complex Type) (§3.3.4).

Validation Rule: Validation Root Valid (ID/IDREF)

For an element information item which is the validation root to be valid all of the following must be true: 1. There must be no ID/IDREF binding in the item's ID/IDREF table whose binding is the empty set. 2. There must be no ID/IDREF binding in the item's ID/IDREF table whose binding has more than one member.

See ID/IDREF Table (§3.15.5) for the definition of ID/IDREF binding.

NOTE: The first clause above applies when there is a reference to an undefined ID. The second applies when there is a multiply-defined ID. They are separated out to ensure that distinct error codes (see Outcome Tabulations (normative) (§C)) are associated with these two cases.

NOTE: Although this rule applies at the validation root, in practice processors, particularly streaming processors, may wish to detect and signal the clause 2 case as it arises.

NOTE: This reconstruction of [XML 1.0 (Second Edition)]'s ID/IDREF functionality is imperfect in that if the validation root is not the document element of an XML document, the results will not necessarily be the same as those a validating parser would give were the
document to have a DTD with equivalent declarations.

Validation Rule: Schema–Validity Assessment (Element)
The schema–validity assessment of an element information item depends on its ·validation· and the ·assessment· of its element information item children and associated attribute information items, if any.

So for an element information item's schema–validity to be assessed all of the following must be true:

1. One of the following must be true:
   1.1 All of the following must be true:
      1.1.1 A non–absent element declaration must be known for it, because one of the following is true:
      1.1.1.1 A declaration was stipulated by the processor (see Assessing Schema–Validity (§5.2)).
      1.1.1.2 A declaration has been established as its ·context–determined declaration·.
      1.1.1.3 All of the following must be true:
         1.1.1.3.1 Its ·context–determined declaration· is not skip.
         1.1.1.3.2 Its [local name] and [namespace name] resolve to an element declaration as defined by QName resolution (Instance) (§3.15.4).
      1.1.2 Its ·validity· with respect to that declaration must have been evaluated as per Element Locally Valid (Element) (§3.3.4). If that evaluation involved the evaluation of Element Locally Valid (Type) (§3.3.4), clause 1 thereof must be satisfied.
      1.2 All of the following must be true:
         1.2.1 A non–absent type definition is known for it because one of the following is true:
         1.2.1.1 A type definition was stipulated by the processor (see Assessing Schema–Validity (§5.2)).
         1.2.1.2 All of the following must be true:
            1.2.1.2.1 There is an attribute information item among the element information item's [attributes] whose [namespace name] is identical to http://www.w3.org/2001/XMLSchema-instance and whose [local name] is type.
            1.2.1.2.2 The ·normalized value· of that attribute information item is ·valid· with respect to the built–in QName simple type, as defined by String Valid (§3.14.4).
            1.2.1.2.3 The [local name] and [namespace name] (as defined in QName Interpretation (§3.15.3)), of the ·actual value· of that attribute information item resolve to a type definition, as defined in QName resolution (Instance) (§3.15.4).—[Definition:] call this type definition the local type definition.
         1.2.2 The element information item's ·validity· with respect to the ·local type definition· (if present and validly derived) or the processor–stipulated type definition (if no ·local type definition· is present) has been evaluated as per Element Locally Valid (Type) (§3.3.4).

2. The schema–validity of all the element information items among its [children] has been assessed as per Schema–Validity Assessment (Element) (§3.3.4), and the schema–validity of all the attribute information items among its [attributes] has been assessed as per Schema–Validity Assessment (Attribute) (§3.2.4).

[Definition:] If either case of clause 1 above holds, the element information item has been strictly assessed.

If the item cannot be strictly assessed, because neither clause 1.1 nor clause 1.2 above are satisfied, [Definition:] an element information item's schema validity may be laxly assessed if its ·context–determined declaration· is not skip by ·validating· with respect to the ·ur–type definition· as per Element Locally Valid (Type) (§3.3.4).

NOTE: In general if clause 1.1 above holds clause 1.2 does not, and vice versa. When an xsi:type [attribute] is involved, however, clause 1.2 takes precedence, as is made clear in Element Locally Valid (Element) (§3.3.4).

NOTE: The [name] and [target namespace] properties are not mentioned above because they are checked during particle ·validation·, as per Element Sequence Locally Valid (Particle) (§3.9.4).

3.3 Element Declarations 94
3.3.5 Element Declaration Information Set Contributions

Schema Information Set Contribution: Assessment Outcome (Element)
If the schema–validity of an element information item has been assessed as per Schema–Validity Assessment (Element) (§3.3.4), then in the post–schema–validation infoset it has properties as follows: PSVI Contributions for element information items

[validation context]
The nearest ancestor element information item with a [schema information] property (or this element item itself if it has such a property).

[validity]
The appropriate case among the following:1 If it was strictly assessed, then the appropriate case among the following:1.1 If all of the following are true1.1.1.1 clause 1.1 of Schema–Validity Assessment (Element) (§3.3.4) applied and the item was valid; as defined by Element Locally Valid (Element) (§3.3.4);1.1.1.2 clause 1.2 of Schema–Validity Assessment (Element) (§3.3.4) applied and the item was valid; as defined by Element Locally Valid (Type) (§3.3.4).1.1.2 Neither its [children] nor its [attributes] contains an information item (element or attribute respectively) whose [validity] is invalid.1.1.3 Neither its [children] nor its [attributes] contains an information item (element or attribute respectively) whose [validity] is unknown. , then valid;1.2 otherwise invalid.2 otherwise notKnown.

[validation attempted]
The appropriate case among the following:1 If it was strictly assessed and neither its [children] nor its [attributes] contains an information item (element or attribute respectively) whose [validation attempted] is not full, then full;2 If it was not strictly assessed, and neither its [children] nor its [attributes] contains an information item (element or attribute respectively) whose [validation attempted] is not none, then none;3 otherwise partial.

Schema Information Set Contribution: Validation Failure (Element)
If the local validity, as defined by Element Locally Valid (Element) (§3.3.4) above and/or Element Locally Valid (Type) (§3.3.4) below, of an element information item has been assessed, in the post–schema–validation infoset the item has a property: PSVI Contributions for element information items

[schema error code]
The appropriate case among the following:1 If the item is not valid, then a list. Applications wishing to provide information as to the reason(s) for the validation failure are encouraged to record one or more error codes (see Outcome Tabulations (normative) (§C)) herein 2 otherwise absent.

Schema Information Set Contribution: Element Declaration
If an element information item is valid with respect to an element declaration as per Element Locally Valid (Element) (§3.3.4) then in the post–schema–validation infoset the element information item must, at processor option, have either: PSVI Contributions for element information items

[element declaration]
an item isomorphic to the declaration component itself

or PSVI Contributions for element information items

[nil]
true if clause 3.2 of Element Locally Valid (Element) (§3.3.4) above is satisfied, otherwise false
Schema Information Set Contribution: Element Validated by Type

If an element information item is \textit{valid} with respect to a \textit{type definition}, as per \texttt{Element Locally Valid (Type)} (§3.3.4), in the post-schema-validation infoset the item has a property: \texttt{PSVI Contributions for element information items}

\[\text{[schema normalized value]}\]

The appropriate \texttt{case} among the following: \texttt{1} \textit{If} clause 3.2 of \texttt{Element Locally Valid (Element)} (§3.3.4) and \texttt{Element Default Value (§3.3.5)} above have \textit{not} applied and either the \textit{type definition} is a simple type definition or its \texttt{[content type]} is a simple type definition, \textit{then} the \texttt{normalized value}, of the item as \textit{validated}; \texttt{2} otherwise \textit{absent}.

Furthermore, the item has one of the following alternative sets of properties:

Either \texttt{PSVI Contributions for element information items}

\[\text{[type definition]}\]

An \texttt{item isomorphic}, to the \texttt{type definition} component itself.

\[\text{[member type definition]}\]

If and only if that type definition is a simple type definition with \texttt{[variety]} \texttt{union}, or a complex type declaration whose \texttt{[content type]} is a simple type definition with \texttt{[variety]} \texttt{union}, then an \texttt{item isomorphic} to that member of the union's \texttt{[member type definitions]} which actually \textit{validated}, the element item's \texttt{normalized value}.

or \texttt{PSVI Contributions for element information items}

\[\text{[type definition type]}\]

\texttt{simple or complex}, depending on the \texttt{type definition}.

\[\text{[type definition namespace]}\]

The \texttt{[target namespace]} of the \texttt{type definition}.

\[\text{[type definition anonymous]}\]

\texttt{true} if the \texttt{[name]} of the \texttt{type definition} is \textit{absent}, otherwise \texttt{false}.

\[\text{[type definition name]}\]

The \texttt{[name]} of the \texttt{type definition}, if it is not \textit{absent}. If it is \textit{absent}, schema processors may, but need not, provide a value unique to the definition.

If the \texttt{type definition} is a simple type definition or its \texttt{[content type]} is a simple type definition, and that type definition has \texttt{[variety]} \texttt{union}, then calling \texttt{[Definition:]} that member of the \texttt{[member type definitions]} which actually \textit{validated}, the element item's \texttt{normalized value}, the \texttt{actual member type definition}, there are three additional properties: \texttt{PSVI Contributions for element information items}

\[\text{[member type definition namespace]}\]

The \texttt{[target namespace]} of the \texttt{actual member type definition}.

\[\text{[member type definition anonymous]}\]

\texttt{true} if the \texttt{[name]} of the \texttt{actual member type definition} is \textit{absent}, otherwise \texttt{false}.

\[\text{[member type definition name]}\]

The \texttt{[name]} of the \texttt{actual member type definition}, if it is not \textit{absent}. If it is \textit{absent}, schema processors may, but need not, provide a value unique to the definition.

The first \texttt{(item isomorphic)} alternative above is provided for applications such as query processors which need access to the full range of details about an item's \texttt{assessment}, for example the type hierarchy; the second, for lighter-weight processors for whom representing the significant parts of the type hierarchy as
information items might be a significant burden.

Also, if the declaration has a value constraint, the item has a property: PSVI Contributions for element information items

[schema default]
The canonical lexical representation of the declaration's value constraint value.

Note that if an element is laxly assessed, then the type definition and member type definition properties, or their alternatives, are based on the ur-type definition. Schema Information Set Contribution: Element Default Value
If the local validity, as defined by Element Locally Valid (Element) (§3.3.4) above, of an element information item has been assessed, in the post-schema-validation infoset the item has a property: PSVI Contributions for element information items

[schema specified]
The appropriate case among the following:1 If the item is valid with respect to an element declaration as per Element Locally Valid (Element) (§3.3.4) and the value constraint is present, but clause 3.2 of Element Locally Valid (Element) (§3.3.4) above is not satisfied and the item has no element or character information item [children], then schema. Furthermore, the post-schema-validation infoset has the canonical lexical representation of the value as the item's schema normalized value property.2 otherwise infoset.

3.3.6 Constraints on Element Declaration Schema Components

All element declarations (see Element Declarations (§3.3)) must satisfy the following constraint.

Schema Component Constraint: Element Declaration Properties Correct
All of the following must be true:1 The values of the properties of an element declaration must be as described in the property tableau in The Element Declaration Schema Component (§3.3.1), modulo the impact of Missing Sub-components (§5.3).2 If there is a value constraint, the canonical lexical representation of its value must be valid with respect to the type definition as defined in Element Default Valid (Immediate) (§3.3.6).3 If there is an substitution group affiliation, the type definition of the element declaration must be validly derived from the type definition of the substitution group affiliation, given the value of the substitution group exclusions of the substitution group affiliation, as defined in Type Derivation OK (Complex) (§3.4.6) (if the type definition is complex) or as defined in Type Derivation OK (Simple) (§3.14.6) (if the type definition is simple).4 If the type definition or type definition's content type is or is derived from ID then there must not be a value constraint.

NOTE: The use of ID as a type definition for elements goes beyond XML 1.0, and should be avoided if backwards compatibility is desired.

The following constraints define relations appealed to elsewhere in this specification.

Schema Component Constraint: Element Default Valid (Immediate)
For a string to be a valid default with respect to a type definition the appropriate case among the following must be true:1 If the type definition is a simple type definition, then the string must be valid with respect to that definition as defined by String Valid (§3.14.4).2 If the type definition is a complex type definition, then all of the following must be true:2.1 its content type must be a simple type definition or mixed.2.2 The appropriate case among the following must be true:2.2.1 If the content type is a simple type definition, then the string must be valid with respect to that simple type definition as defined by String Valid (§3.14.4).2.2.2...
If the \{content type\} is \textit{mixed}, then the \{content type\}'s particle must be \textit{emptiable} as defined by Particle Emptiable (§3.9.6). **Schema Component Constraint: Substitution Group OK (Transitive)**

For an element declaration (call it D) together with a blocking constraint (a subset of \{substitution, extension, restriction\}, the value of a \{disallowed substitutions\}) to be validly substitutable for another element declaration (call it C) all of the following must be true:

1. The blocking constraint does not contain substitution.
2. There is a chain of \{substitution group affiliation\}'s from D to C, that is, either D's \{substitution group affiliation\} is C, or D's \{substitution group affiliation\}'s \{substitution group affiliation\} is C, or . . .
3. The set of all \{derivation method\}s involved in the derivation of D's \{type definition\} from C's \{type definition\} does not intersect with the union of the blocking constraint, C's \{prohibited substitutions\} (if C is complex, otherwise the empty set) and the \{prohibited substitutions\} (respectively the empty set) of any intermediate \{type definition\}'s in the derivation of D's \{type definition\} from C's \{type definition\}.

**Schema Component Constraint: Substitution Group**

[Definition:] Every element declaration in the \{element declarations\} of a schema defines a \textit{substitution group}, a subset of those \{element declarations\}, as follows:

1. The element declaration itself is in the group;
2. The group is closed with respect to \{substitution group affiliation\}, that is, if any element declaration in the \{element declarations\} has a \{substitution group affiliation\} in the group, then it is also in the group itself.

### 3.4 Complex Type Definitions

3.4.1 The Complex Type Definition Schema Component

3.4.2 XML Representation of Complex Type Definitions

3.4.3 Constraints on XML Representations of Complex Type Definitions

3.4.4 Complex Type Definition Validation Rules

3.4.5 Complex Type Definition Information Set Contributions

3.4.6 Constraints on Complex Type Definition Schema Components

3.4.7 Built−in Complex Type Definition

Complex Type Definitions provide for:

- Constraining element information items by providing Attribute Declaration (§2.2.2.3)s governing the appearance and content of \[attributes\].
- Constraining element information item \[children\] to be empty, or to conform to a specified element-only or mixed content model, or else constraining the character information item \[children\] to conform to a specified simple type definition.
- Using the mechanisms of Type Definition Hierarchy (§2.2.1.1) to derive a complex type from another simple or complex type.
- Specifying \textit{post−schema−validation infoset contributions} for elements.
- Limiting the ability to derive additional types from a given complex type.
- Controlling the permission to substitute, in an instance, elements of a derived type for elements declared in a content model to be of a given complex type.

Example

```
<xs:complexType name="PurchaseOrderType">
  <xs:sequence>
    <xs:element name="shipTo" type="USAddress"/>
    <xs:element name="billTo" type="USAddress"/>
    <xs:element ref="comment" minOccurs="0"/>
    <xs:element name="items" type="Items"/>
  </xs:sequence>
  <xs:attribute name="orderDate" type="xs:date"/>
</xs:complexType>
```

The XML representation of a complex type definition.
3.4.1 The Complex Type Definition Schema Component

A complex type definition schema component has the following properties:

Schema Component: Complex Type Definition

{name}
  Optional. An NCName as defined by [XML−Namespaces].
{target namespace}
  Either absent, or a namespace name, as defined in [XML−Namespaces].
{base type definition}
  Either a simple type definition or a complex type definition.
{derivation method}
  Either extension or restriction.
{final}
  A subset of {extension, restriction}.
{abstract}
  A boolean
{attribute uses}
  A set of attribute uses.
{attribute wildcard}
  Optional. A wildcard.
{content type}
  One of empty, a simple type definition or a pair consisting of a content model. (I.e. a Particle ($2.2.3.2)) and one of mixed, element−only.
{prohibited substitutions}
  A subset of {extension, restriction}.
{annotations}
  A set of annotations.

Complex types definitions are identified by their {name} and {target namespace}. Except for anonymous complex type definitions (those with no {name}), since type definitions (i.e. both simple and complex type definitions taken together) must be uniquely identified within an XML Schema, no complex type definition can have the same name as another simple or complex type definition. Complex type {name}s and {target namespace}s are provided for reference from instances (see xsi:type ($2.6.1)), and for use in the XML representation of schema components (specifically in <element>). See References to schema components across namespaces ($4.2.3) for the use of component identifiers when importing one schema into another.

NOTE: The {name} of a complex type is not ipso facto the [(local) name] of the element information items validated by that definition. The connection between a name and a type definition is described in Element Declarations ($3.3).

As described in Type Definition Hierarchy ($2.2.1.1), each complex type is derived from a {base type definition} which is itself either a Simple Type Definition ($2.2.1.2) or a Complex Type Definition ($2.2.1.3). {derivation method} specifies the means of derivation as either extension or restriction (see Type Definition Hierarchy ($2.2.1.1)).

A complex type with an empty specification for {final} can be used as a {base type definition} for other types derived by either of extension or restriction; the explicit values extension, and restriction prevent further derivations by extension and restriction respectively. If all values are specified, then [Definition:] the complex type is said to be final, because no further derivations are possible. Finality is not inherited, that is, a
type definition derived by restriction from a type definition which is final for extension is not itself, in the absence of any explicit final attribute of its own, final for anything.

Complex types for which {abstract} is true must not be used as the {type definition} for the -validation- of element information items. It follows that they must not be referenced from an xsi:type ($2.6.1) attribute in an instance document. Abstract complex types can be used as {base type definition}s, or even as the {type definition}s of element declarations, provided in every case a concrete derived type definition is used for -validation-, either via xsi:type ($2.6.1) or the operation of a substitution group.

{attribute uses} are a set of attribute uses. See Element Locally Valid (Complex Type) ($3.4.4) and Attribute Locally Valid ($3.2.4) for details of attribute -validation-.

{attribute wildcard}s provide a more flexible specification for -validation- of attributes not explicitly included in {attribute uses}. Informally, the specific values of {attribute wildcard} are interpreted as follows:

- any: [attributes] can include attributes with any qualified or unqualified name.
- a set whose members are either namespace names or -absent-: [attributes] can include any attribute(s) from the specified namespace(s). If -absent- is included in the set, then any unqualified attributes are (also) allowed.
- 'not' and a namespace name: [attributes] cannot include attributes from the specified namespace.
- 'not' and -absent-: [attributes] cannot include unqualified attributes.

See Element Locally Valid (Complex Type) ($3.4.4) and Wildcard allows Namespace Name ($3.10.4) for formal details of attribute wildcard -validation-.

{content type} determines the -validation- of [children] of element information items. Informally:

- A {content type} with the distinguished value empty -validates- elements with no character or element information item [children].
- A {content type} which is a Simple Type Definition ($2.2.1.2) -validates- elements with character-only [children].
- An element-only {content type} -validates- elements with [children] that conform to the supplied -content model-.
- A mixed {content type} -validates- elements whose element [children] (i.e. specifically ignoring other [children] such as character information items) conform to the supplied -content model-.

{prohibited substitutions} determine whether an element declaration appearing in a -content model- is prevented from additionally -validating- element items with an xsi:type ($2.6.1) attribute that identifies a complex type definition derived by extension or restriction from this definition, or element items in a substitution group whose type definition is similarly derived: If {prohibited substitutions} is empty, then all such substitutions are allowed, otherwise, the derivation method(s) it names are disallowed.

See Annotations ($3.13) for information on the role of the {annotations} property.

3.4.2 XML Representation of Complex Type Definitions

The XML representation for a complex type definition schema component is a <complexType> element information item.

The XML representation for complex type definitions with a simple type definition {content type} is significantly different from that of those with other {content type}s, and this is reflected in the presentation
below, which displays first the elements involved in the first case, then those for the second. The property
mapping is shown once for each case.

XML Representation Summary: `complexType` Element Information Item

```xml
<complexType
    abstract = boolean : false
    block = (#all | List of (extension | restriction))
    final = (#all | List of (extension | restriction))
    id = ID
    mixed = boolean : false
    name = NCName
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, (simpleContent | complexContent | ((group | all | choice | sequence)?, ((attribute | attributeGroup)*) , anyAttribute?)))
</complexType>
```

Whichever alternative for the content of `<complexType>` is chosen, the following property mappings apply:

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{name}</code></td>
<td>The <em>actual value</em> of the <code>name</code> attribute if present, otherwise <em>absent</em>.</td>
</tr>
<tr>
<td><code>{target namespace}</code></td>
<td>The <em>actual value</em> of the <code>targetNamespace</code> attribute of the <code>&lt;schema&gt;</code> ancestor element information item if present, otherwise <em>absent</em>.</td>
</tr>
<tr>
<td><code>{abstract}</code></td>
<td>The <em>actual value</em> of the <code>abstract</code> attribute, if present, otherwise <em>false</em>.</td>
</tr>
<tr>
<td><code>{prohibited substitutions}</code></td>
<td>A set corresponding to the <em>actual value</em> of the <code>block</code> attribute, if present, otherwise on the <em>actual value</em> of the <code>blockDefault</code> attribute of the ancestor <code>&lt;schema&gt;</code> element information item, if present, otherwise on the empty string. Call this the EBV (for effective block value). Then the value of this property is the appropriate <em>case</em> among the following:1 If the EBV is the empty string, then the empty set; 2 If the EBV is #all, then <code>{extension, restriction}</code>; 3 otherwise a set with members drawn from the set above, each being present or absent depending on whether the <em>actual value</em> (which is a list) contains an equivalently named item. NOTE: Although the <code>blockDefault</code> attribute of <code>&lt;schema&gt;</code> may include values other than restriction or extension, those values are ignored in the determination of <code>{prohibited substitutions}</code> for complex type definitions (they are used elsewhere).</td>
</tr>
<tr>
<td><code>{final}</code></td>
<td>As for <code>{prohibited substitutions}</code> above, but using the <code>final</code> and <code>finalDefault</code> attributes in place of the block and blockDefault attributes.</td>
</tr>
<tr>
<td><code>{annotations}</code></td>
<td>The annotations corresponding to the <code>&lt;annotation&gt;</code> element information item in the [children], if present, in the <code>&lt;simpleContent&gt;</code> and <code>&lt;complexContent&gt;</code> [children], if present, and in their <code>&lt;restriction&gt;</code> and <code>&lt;extension&gt;</code> [children], if</td>
</tr>
</tbody>
</table>
present, otherwise absent.

When the <simpleContent> alternative is chosen, the following elements are relevant, and the remaining property mappings are as below. Note that either <restriction> or <extension> must be chosen as the content of <simpleContent>.

<simpleContent
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</simpleContent>

<restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive | totalDigits | fractionDigits | length | minLength | maxLength | enumeration | whiteSpace | pattern)*),
  ((attribute | attributeGroup)*, anyAttribute?)
</restriction>

<extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((attribute | attributeGroup)*, anyAttribute?))
</extension>

<attributeGroup
  id = ID
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</attributeGroup>

<anyAttribute
  id = ID
  namespace = (##any | ##other ) | List of (anyURI | (#targetNamespace | ##local)) ) : ##any
  processContents = (lax | skip | strict) : strict
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</anyAttribute>

**Complex Type Definition with simple content Schema Component**
{base type definition} The type definition -resolved- to by the -actual value- of the base [attribute].

{derivation method} If the <restriction> alternative is chosen, then restriction, otherwise (the <extension> alternative is chosen) extension.

{attribute uses} A union of sets of attribute uses as follows:1 The set of attribute uses corresponding to the <attribute> [children], if any.2 The [attribute uses] of the attribute groups -resolved- to by the -actual value-s of the ref [attribute] of the <attributeGroup> [children], if any.3 if the type definition -resolved- to by the -actual value- of the base [attribute] is a complex type definition, the [attribute uses] of that type definition, unless the <restriction> alternative is chosen, is in which case some members of that type definition’s [attribute uses] may not be included, namely those whose [attribute declaration]’s [name] and [target namespace] are the same as one of the following:3.1 the [name] and [target namespace] of the [attribute declaration] of an attribute use in the set per clause 1 or clause 2 above;3.2 what would have been the [name] and [target namespace] of the [attribute declaration] of an attribute use in the set per clause 1 above but for the -actual value- of the use [attribute] of the relevant <attribute> among the [children] of <restriction> being prohibited.

{attribute wildcard} [Definition:] Let the local wildcard be defined as the appropriate case among the following:1 If there is an <anyAttribute> present, then a wildcard based on the -actual value-s of the namespace and processContents [attributes] and the <annotation> [children], exactly as for the wildcard corresponding to an <any> element as set out in XML Representation of Wildcard Schema Components (§3.10.2);2 otherwise absent. [Definition:] Let the complete wildcard be defined as the appropriate case among the following:1 If there are no <attributeGroup> [children] corresponding to attribute groups with non-absent {attribute wildcard}s, then the local wildcard.2 If there are one or more <attributeGroup> [children] corresponding to attribute groups with non-absent {attribute wildcard}s, then the appropriate case among the following:2.1 If there is an <anyAttribute> present, then a wildcard whose [process contents] and [annotation] are those of the local wildcard, and whose [namespace constraint] is the intensional intersection of the [namespace constraint] of the local wildcard, and of the [namespace constraint]s of all the non-absent {attribute wildcard}s of the attribute groups corresponding to the <attributeGroup> [children], as defined in Attribute Wildcard Intersection (§3.10.6).2.2 If there is no <anyAttribute> present, then a wildcard whose properties are as follows:

{process contents} The [process contents] of the first non-absent {attribute wildcard} of an attribute group among the attribute groups corresponding to the <attributeGroup> [children].

{namespace constraint} The intensional intersection of the [namespace constraint]s of all the non-absent {attribute wildcard}s of the attribute groups corresponding to the <attributeGroup> [children], as defined in Attribute Wildcard Intersection (§3.10.6).

{annotation}
The value is then determined by the appropriate case among the following: 1 If the `<restriction>` alternative is chosen, then the complete wildcard; 2 If the `<extension>` alternative is chosen, then [Definition:] let the base wildcard be defined as the appropriate case among the following: 2.1 If the type definition resolved to by the actual value of the base [attribute] is a complex type definition with an {attribute wildcard}, then that {attribute wildcard}. 2.2 otherwise absent. The value is then determined by the appropriate case among the following: 2.1 If the base wildcard is non-absent, then the appropriate case among the following: 2.1.1 If the complete wildcard is absent, then the base wildcard. 2.1.2 otherwise a wildcard whose {process contents} and {annotation} are those of the complete wildcard, and whose {namespace constraint} is the intensional union of the {namespace constraint} of the effective wildcard and of the base wildcard, as defined in Attribute Wildcard Union (§3.10.6).

1 if the type definition resolved to by the actual value of the base [attribute] is a complex type definition (whose own {content type} must be a simple type definition, see below) and the `<restriction>` alternative is chosen, then starting from either 1.1 the simple type definition corresponding to the `<simpleType>` among the [children] of `<restriction>` if there is one; 1.2 otherwise `<restriction>` has no `<simpleType>` among its [children], the simple type definition which is the {content type} of the type definition resolved to by the actual value of the base [attribute] a simple type definition which restricts that simple type definition with a set of facet components corresponding to the appropriate element information items among the `<restriction>`’s [children] (i.e. those which specify facets, if any), as defined in Simple Type Restriction (Facets) (§3.14.3); 2 otherwise if the type definition resolved to by the actual value of the base [attribute] is a complex type definition (whose own {content type} must be a simple type definition, see below) and the `<extension>` alternative is chosen, then the {content type} of that complex type definition; 3 otherwise (the type definition resolved to by the actual value of the base [attribute] is a simple type definition and the `<extension>` alternative is chosen), then that simple type definition.

When the `<complexContent>` alternative is chosen, the following elements are relevant (as are the `<attributeGroup>` and `<anyAttribute>` elements, not repeated here), and the additional property mappings are as below. Note that either `<restriction>` or `<extension>` must be chosen as the content of `<complexContent>`. The property mappings below are also used in the case where the third alternative (neither `<simpleContent>` nor `<complexContent>`) is chosen. This case is understood as shorthand for complex content restricting the ur-type definition, and the details of the mappings should be modified as necessary.

```xml
<complexContent
  id = ID
  mixed = boolean
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</complexContent>

<restriction
  base = QName
```
Complex Type Definition with complex content Schema Component

**Property**

*base type definition*  
The type definition resolved to by the actual value of the base [attribute]

*derivation method*  
If the <restriction> alternative is chosen, then restriction, otherwise (the <extension> alternative is chosen) extension.

*attribute uses*  
A union of sets of attribute uses as follows:1 The set of attribute uses corresponding to the <attribute> [children], if any.2 The {attribute uses} of the attribute groups resolved to by the actual values of the ref [attribute] of the <attributeGroup> [children], if any.3 The {attribute uses} of the type definition resolved to by the actual value of the base [attribute], unless the <restriction> alternative is chosen, in which case some members of that type definition's {attribute uses} may not be included, namely those whose {attribute declaration}'s {name} and {target namespace} are the same as one of the following:3.1 The {name} and {target namespace} of the {attribute declaration} of an attribute use in the set per clause 1 or clause 2 above;3.2 what would have been the {name} and {target namespace} of the {attribute declaration} of an attribute use in the set per clause 1 above but for the actual value of the use [attribute] of the relevant <attribute> among the [children] of <restriction> being prohibited.

*attribute wildcard*  
As above for the <simpleContent> alternative.

*content type*  
The appropriate case among the following:1 If the <restriction> alternative is chosen, then the appropriate case among the following:1.1 If one of the following is true:1.1.1 There is no <group>, <all>, <choice> or <sequence> among the [children];1.1.2 There is an <all> or <sequence> among the [children] with no [children] of its own excluding <annotation>;1.1.3 There is a <choice> among the [children] with no [children] of its own excluding <annotation> whose minOccurs [attribute] has the actual value: 0, then empty;1.2 otherwise a pair consisting of1.2.1 the appropriate case among the following:1.2.1.1 If the mixed [attribute] is present on <complexContent>, then mixed if its actual value is true, otherwise elementOnly;1.2.1.2 If the mixed [attribute] is present on <complexType> and its actual value is true, then mixed;1.2.1.3 otherwise elementOnly.1.2.2 The particle corresponding to the <all>, <choice>, <group> or <sequence> among the [children].2 If the <extension> alternative is chosen, then [Definition:] let the explicit content be empty if any of the sub-clauses of clause 1.1 above applies, otherwise the particle
corresponding to the `<all>`, `<choice>`, `<group>` or `<sequence>` among the `[children]`, and then take the appropriate **case** among the following:

2.1 If the **explicit content** is **empty**, then the `[content type]` of the type definition `-resolved-` to by the `-actual value-` of the `base` `[attribute]`

2.2 If the type definition `-resolved-` to by the `-actual value-` of the `base` `[attribute]` has a `[content type]` of **empty**, then a pair of mixed or `elementOnly` (determined as per clause 1.2.1 above) and the `-explicit content-` itself;

2.3 otherwise a pair of mixed or `elementOnly` (determined as per clause 1.2.1 above) and a particle whose properties are as follows:

- **[min occurs]**
  - 1

- **[max occurs]**
  - 1

- **[term]**

  A model group whose `[compositor]` is `sequence` and whose `[particles]` are the particle of the `[content type]` of the type definition `-resolved-` to by the `-actual value-` of the `base` `[attribute]` followed by the `-explicit content-`.

**NOTE:** Aside from the simple coherence requirements enforced above, constraining type definitions identified as restrictions to actually be restrictions, that is, to `-validate-` a subset of the items which are `-validated-` by their base type definition, is enforced in Constraints on Complex Type Definition Schema Components (§3.4.6).

**NOTE:** The **only** substantive function of the value **prohibited** for the `use` attribute of an `<attribute>` is in establishing the correspondence between a complex type defined by restriction and its XML representation. It serves to prevent inheritance of an identically named attribute use from the `{base type definition}`. Such an `<attribute>` does not correspond to any component, and hence there is no interaction with either explicit or inherited wildcards in the operation of Complex Type Definition Validation Rules (§3.4.4) or Constraints on Complex Type Definition Schema Components (§3.4.6).

Careful consideration of the above concrete syntax reveals that a type definition need consist of no more than a name, i.e. that `<complexType name="anyThing"/>` is allowed.

**Example**

```
<xs:complexType name="length1">
   <xs:simpleContent>
      <xs:extension base="xs:nonNegativeInteger">
         <xs:attribute name="unit" type="xs:NMTOKEN"/>
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>

<xs:element name="width" type="length1"/>
   <width unit="cm">25</width>

<xs:complexType name="length2">
   <xs:complexContent>
      <xs:restriction base="xs:anyType">
         <xs:sequence>
            <xs:element name="size" type="xs:nonPositiveInteger"/>
            <xs:element name="unit" type="xs:NMTOKEN"/>
         </xs:sequence>
      </xs:restriction>
   </xs:complexContent>
</xs:complexType>
```
Three approaches to defining a type for length: one with character data content constrained by reference to a built-in datatype, and one attribute, the other two using two elements. length3 is the abbreviated alternative to length2: they correspond to identical type definition components.

Example

```xml
<xs:complexType name="length3">
  <xs:sequence>
    <xs:element name="size" type="xs:non-positive-integer"/>
    <xs:element name="unit" type="xs:NMTOKEN"/>
  </xs:sequence>
</xs:complexType>
```

A type definition for personal names, and a definition derived by extension which adds a single element; an element declaration referencing the derived definition, and a valid instance thereof.

Example

```xml
<xs:complexType name="simpleName">
  <xs:complexContent>
    <xs:restriction base="personName">
      <xs:sequence>
        <xs:element name="forename" minOccurs="1" maxOccurs="1"/>
        <xs:element name="surname"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```
A simplified type definition derived from the base type from the previous example by restriction, eliminating one optional daughter and fixing another to occur exactly once; an element declared by reference to it, and a valid instance thereof. Example

```
<xs:complexType name="paraType" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element ref="emph"/>
    <xs:element ref="strong"/>
  </xs:choice>
  <xs:attribute name="version" type="xs:number"/>
</xs:complexType>
```

A further illustration of the abbreviated form, with the mixed attribute appearing on complexType itself.

### 3.4.3 Constraints on XML Representations of Complex Type Definitions

**Schema Representation Constraint: Complex Type Definition Representation OK**

In addition to the conditions imposed on `<complexType>` element information items by the schema for schemas, all of the following must be true: 1. If the `<complexContent>` alternative is chosen, the type definition resolved to by the actual value of the base attribute must be a complex type definition; 2. If the `<simpleContent>` alternative is chosen, the type definition resolved to by the actual value of the base attribute must be either a complex type definition whose {content type} is a simple type definition or, only if the `<extension>` alternative is also chosen, a simple type definition; 3. The corresponding complex type definition component must satisfy the conditions set out in Constraints on Complex Type Definition Schema Components (§3.4.6); 4. If clause 2.1 or clause 2.2 in the correspondence specification above for {attribute wildcard} is satisfied, the intensional intersection must be expressible, as defined in Attribute Wildcard Intersection (§3.10.6).

### 3.4.4 Complex Type Definition Validation Rules

**Validation Rule: Element Locally Valid (Complex Type)**

For an element information item to be locally valid with respect to a complex type definition all of the following must be true: 1. `{abstract}` is false. 2. If clause 3.2 of Element Locally Valid (Element) (§3.3.4) did not apply, then the appropriate case among the following must be true: 2.1 If the `{content type}` is empty, then the element information item has no character or element information item [children]. 2.2 If the `{content type}` is a simple type definition, then the element information item has no element information item [children]. 2.3 If the `{content type}` is element-only, then the element information item has no character information item [children] other than those whose [character code] is defined as a white space in [XML 1.0 (Second Edition)]. 2.4 If the `{content type}` is element-only or mixed, then the sequence of the element information item's element information item [children], if any, taken in order, is valid, with respect to the `{content type}`'s particle, as defined in Element Sequence Locally Valid (Particle).
For each attribute information item in the element information item's [attributes] excepting those whose [namespace name] is identical to http://www.w3.org/2001/XMLSchema-instance and whose [local name] is one of type, nil, schemaLocation or noNamespaceSchemaLocation, the appropriate case among the following must be true:

3.1 If there is among the [attribute uses] an attribute use with an [attribute declaration] whose [name] matches the attribute information item's [local name] and whose [target namespace] is identical to the attribute information item's [namespace name] (where an -absent- [target namespace] is taken to be identical to a [namespace name] with no value), then the attribute information must be ·valid· with respect to that attribute use as per Attribute Locally Valid (Use) (§3.5.4). In this case the [attribute declaration] of that attribute use is the ·context–determined declaration· for the attribute information item with respect to Schema–Validity Assessment (Attribute) (§3.2.4) and Assessment Outcome (Attribute) (§3.2.5).

3.2 otherwise all of the following must be true:

3.2.1 There must be an [attribute wildcard].

3.2.2 The attribute information item must be ·valid· with respect to it as defined in Item Valid (Wildcard) (§3.10.4).

4 The [attribute declaration] of each attribute use in the [attribute uses] whose [required] is true matches one of the attribute information items in the element information item's [attributes] as per clause 3.1 above.

5 Let [Definition:] the wild IDs be the set of all attribute information item to which clause 3.2 applied and whose ·validation· resulted in a ·context–determined declaration· of mustFind or no ·context–determined declaration· at all, and whose [local name] and [namespace name] resolve (as defined by QName resolution (Instance) (§3.15.4)) to an attribute declaration whose {type definition} is or is derived from ID. Then all of the following must be true:

5.1 There must be no more than one item in ·wild IDs·.

5.2 If ·wild IDs· is non–empty, there must not be any attribute uses among the [attribute uses] whose [attribute declaration]’s {type definition} is or is derived from ID.

NOTE: This clause serves to ensure that even via attribute wildcards no element has more than one attribute of type ID, and that even when an element legitimately lacks a declared attribute of type ID, a wildcard–validated attribute must not supply it. That is, if an element has a type whose attribute declarations include one of type ID, it either has that attribute or no attribute of type ID.

NOTE: When an [attribute wildcard] is present, this does not introduce any ambiguity with respect to how attribute information items for which an attribute use is present amongst the [attribute uses] whose name and target namespace match are ·assessed·. In such cases the attribute use always takes precedence, and the ·assessment· of such items stands or falls entirely on the basis of the attribute use and its [attribute declaration]. This follows from the details of clause 3.

### 3.4.5 Complex Type Definition Information Set Contributions

#### Schema Information Set Contribution: Attribute Default Value

For each attribute use in the [attribute uses] whose [required] is false and whose [value constraint] is not -absent·, but whose [attribute declaration] does not match one of the attribute information items in the element information item's {attributes} as per clause 3.1 of Element Locally Valid (Complex Type) (§3.4.4) above, the post–schema–validation infoset has an attribute information item whose properties are as below added to the [attributes] of the element information item.

**[local name]**

The [attribute declaration]’s {name}.

**[namespace name]**

The [attribute declaration]’s {target namespace}.

**[schema normalized value]**

The canonical lexical representation of the {value constraint} value.

**[schema default]**

3.4 Complex Type Definitions 109
The canonical lexical representation of the \{value constraint\} value.

\textbf{validation context}

The nearest ancestor element information item with a \{schema information\} property.

\textbf{validity}

valid.

\textbf{validation attempted}

full.

\textbf{schema specified}

schema.

The added items should also either have \{type definition\} (and \{member type definition\} if appropriate) properties, or their lighter−weight alternatives, as specified in Attribute Validated by Type (§3.2.5).

### 3.4.6 Constraints on Complex Type Definition Schema Components

All complex type definitions (see Complex Type Definitions (§3.4)) must satisfy the following constraints.

\textbf{Schema Component Constraint: Complex Type Definition Properties Correct}

All of the following must be true: 1 The values of the properties of a complex type definition must be as described in the property tableau in The Complex Type Definition Schema Component (§3.4.1), modulo the impact of Missing Sub−components (§5.3). 2 If the \{base type definition\} is a simple type definition, the \{derivation method\} must be \textit{extension}. 3 Circular definitions are disallowed, except for the \textit{ur−type definition}. That is, it must be possible to reach the \textit{ur−type definition} by repeatedly following the \{base type definition\}. 4 Two distinct attribute declarations in the \{attribute uses\} must not have identical \{name\}s and \{target namespace\}s. 5 Two distinct attribute declarations in the \{attribute uses\} must not have \{type definition\}s which are or are derived from ID.

\textbf{Schema Component Constraint: Derivation Valid (Extension)}

If the \{derivation method\} is \textit{extension}, the appropriate case among the following must be true: 1 If the \{base type definition\} is a complex type definition, then all of the following must be true: 1.1 The \{final\} of the \{base type definition\} must not contain \textit{extension}. 1.2 Its \{attribute uses\} must be a subset of the \{attribute uses\} of the complex type definition itself, that is, for every attribute use in the \{attribute uses\} of the \{base type definition\}, there must be an attribute use in the \{attribute uses\} of the complex type definition itself whose \{attribute declaration\} has the same \{name\}, \{target namespace\} and \{type definition\} as its attribute declaration. 1.3 If it has an \{attribute wildcard\}, the complex type definition must also have one, and the base type definition's \{attribute wildcard\}'s \{namespace constraint\} must be a subset of the complex type definition's \{attribute wildcard\}'s \{namespace constraint\}, as defined by Wildcard Subset (§3.10.6). 1.4 One of the following must be true: 1.4.1 The \{content type\} of the \{base type definition\} and the \{content type\} of the complex type definition itself must be the same simple type definition. 1.4.2 All of the following must be true: 1.4.2.1 The \{content type\} of the complex type definition itself must specify a particle. 1.4.2.2 One of the following must be true: 1.4.2.2.1 The \{content type\} of the \{base type definition\} must be \textit{empty}. 1.4.2.2.2 All of the following must be true: 1.4.2.2.2.1 Both \{content type\}s must be \textit{mixed} or both must be \textit{element−only}. 1.4.2.2.2.2 The particle of the complex type definition must be a \textit{valid extension} of the \{base type definition\}'s particle, as defined in Particle Valid (Extension) (§3.9.6). 1.5 It must in principle be possible to derive the complex type definition in two steps, the first an extension and the second a restriction (possibly vacuous), from that type definition among its ancestors whose \{base type definition\} is the \textit{ur−type definition}.

\textbf{NOTE:} This requirement ensures that nothing removed by a restriction is subsequently added back by an extension. It is trivial to check if the extension in question is the only extension in its derivation, or if there are no restrictions bar the first from the \textit{ur−type definition}.
Constructing the intermediate type definition to check this constraint is straightforward: simply re-order the derivation to put all the extension steps first, then collapse them into a single extension. If the resulting definition can be the basis for a valid restriction to the desired definition, the constraint is satisfied.

2 If the `{base type definition}` is a simple type definition, then all of the following must be true:

- The `{content type}` must be the same simple type definition.
- The `{final}` of the `{base type definition}` must not contain extension.

[Definition:] If this constraint Derivation Valid (Extension) (§3.4.6) holds of a complex type definition, it is a valid extension of its `{base type definition}`. Schema Component Constraint: Derivation Valid (Restriction, Complex)

If the `{derivation method}` is restriction all of the following must be true:

- The `{base type definition}` must be a complex type definition whose `{final}` does not contain restriction.
- For each attribute use (call this R) in the `{attribute uses}` the appropriate case among the following must be true:
  - If there is an attribute use in the `{attribute uses}` of the `{base type definition}` (call this B) whose `{attribute declaration}` has the same `{name}` and `{target namespace}`, then all of the following must be true:
    - 2.1.1 One of the following must be true:
      - 2.1.1.1 B's `{required}` is false.
      - 2.1.1.2 R's `{required}` is true.
    - 2.1.2 R's `{attribute declaration}`'s `{type definition}` must be validly derived from B's `{type definition}` given the empty set as defined in Type Derivation OK (Simple) (§3.14.6).
    - 2.1.3 Let the effective value constraint of an attribute use be its `{value constraint}`, if present, otherwise its `{attribute declaration}`'s `{value constraint}`. Then one of the following must be true:
      - 2.1.3.1 B's `{effective value constraint}` is fixed.
      - 2.1.3.2 R's `{effective value constraint}` is fixed.

- For each attribute use in a complex type definition whose `{final}` does not contain restriction all of the following must be true:
  - 2.2 The `{final}` of the `{base type definition}` must not contain an attribute use of the complex type definition itself whose `{required}` is true.

NOTE: To restrict a complex type definition with a simple base type definition to empty, use a simple type definition with a fixed value of the empty string; this preserves the type information.

The following constraint defines a relation appealed to elsewhere in this specification.

Schema Component Constraint: Type Derivation OK (Complex)

For a complex type definition (call it D, for derived) to be validly derived from a type definition (call this B,
for base) given a subset of \{extension, restriction\} all of the following must be true:

1. If B and D are not the same type definition, then the \{derivation method\} of D must not be in the subset.
2. One of the following must be true:
   2.1 B and D must be the same type definition.
   2.2 If D's \{base type definition\} is complex, then it must be validly derived from B given the subset as defined by this constraint.
   2.3 If D's \{base type definition\} is simple, then it must be validly derived from B given the subset as defined in Type Derivation OK (Simple) (§3.14.6).

NOTE: This constraint is used to check that when someone uses a type in a context where another type was expected (either via xsi:type or substitution groups), that the type used is actually derived from the expected type, and that that derivation does not involve a form of derivation which was ruled out by the expected type.

### 3.4.7 Built-in Complex Type Definition

There is a complex type definition nearly equivalent to the ur-type definition: present in every schema by definition. It has the following properties:

<table>
<thead>
<tr>
<th>Complex Type Definition of the Ur-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
</tr>
<tr>
<td>{name}</td>
</tr>
<tr>
<td>{target namespace}</td>
</tr>
<tr>
<td>{base type definition}</td>
</tr>
<tr>
<td>{derivation method}</td>
</tr>
</tbody>
</table>

A pair consisting of mixed and a particle with the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{min occurs}</td>
<td>1</td>
</tr>
<tr>
<td>{max occurs}</td>
<td>1</td>
</tr>
</tbody>
</table>

A model group with the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{compositor}</td>
<td>sequence</td>
</tr>
</tbody>
</table>

A list containing one particle with the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{min occurs}</td>
<td>0</td>
</tr>
<tr>
<td>{max occurs}</td>
<td>unbounded</td>
</tr>
</tbody>
</table>

A wildcard with an any namespace constraint:
The mixed content specification together with the unconstrained wildcard content model and attribute specification produce the defining property for the _ur-type definition_, namely that every complex type definition is (eventually) a restriction of the _ur-type definition_: its permissions and requirements are the least restrictive possible.

**NOTE:** This specification does not provide an inventory of built-in complex type definitions for use in user schemas. A preliminary library of complex type definitions is available which includes both mathematical (e.g. `rational`) and utility (e.g. `array`) type definitions. In particular, there is a `text` type definition which is recommended for use as the type definition in element declarations intended for general text content, as it makes sensible provision for various aspects of internationalization. For more details, see the schema document for the type library at its namespace name:

[http://www.w3.org/2001/03/XMLSchema/TypeLibrary.xsd](http://www.w3.org/2001/03/XMLSchema/TypeLibrary.xsd).

### 3.5 AttributeUses

3.5.1 The Attribute Use Schema Component

An attribute use is a utility component which controls the occurrence and defaulting behavior of attribute declarations. It plays the same role for attribute declarations in complex types that particles play for element declarations.

**Example**

```xml
<xs:complexType>
  ...<xs:attribute ref="xml:lang" use="required"/>
  <xs:attribute ref="xml:space" default="preserve"/>
  <xs:attribute name="version" type="xs:number" fixed="1.0"/>
</xs:complexType>
```

XML representations which all involve attribute uses, illustrating some of the possibilities for controlling occurrence.

#### 3.5.1 The Attribute Use Schema Component

The attribute use schema component has the following properties:

Schema Component: **Attribute Use**
A boolean.

An attribute declaration.

Optional. A pair consisting of a value and one of default, fixed.

{required} determines whether this use of an attribute declaration requires an appropriate attribute information item to be present, or merely allows it.

{attribute declaration} provides the attribute declaration itself, which will in turn determine the simple type definition used.

{value constraint} allows for local specification of a default or fixed value. This must be consistent with that of the {attribute declaration}, in that if the {attribute declaration} specifies a fixed value, the only allowed {value constraint} is the same fixed value.

**3.5.2 XML Representation of Attribute Use Components**

Attribute uses correspond to all uses of <attribute> which allow a use attribute. These in turn correspond to two components in each case, an attribute use and its {attribute declaration} (although note the latter is not new when the attribute use is a reference to a top-level attribute declaration). The appropriate mapping is described in XML Representation of Attribute Declaration Schema Components (§3.2.2).

**3.5.3 Constraints on XML Representations of Attribute Uses**

None as such.

**3.5.4 Attribute Use Validation Rules**

Validation Rule: Attribute Locally Valid (Use)
For an attribute information item to be valid with respect to an attribute use its normalized value must match the canonical lexical representation of the attribute use's {value constraint} value, if it is present and fixed.

**3.5.5 Attribute Use Information Set Contributions**

None as such.

**3.5.6 Constraints on Attribute Use Schema Components**

All attribute uses (see AttributeUses (§3.5)) must satisfy the following constraints.

Schema Component Constraint: Attribute Use Correct
All of the following must be true: 1 The values of the properties of an attribute use must be as described in the property tableau in The Attribute Use Schema Component (§3.5.1), modulo the impact of Missing Sub-components (§5.3). 2 If the {attribute declaration} has a fixed {value constraint}, then if the attribute use itself has a {value constraint}, it must also be fixed and its value must match that of the {attribute declaration}'s {value constraint}.
3.6 Attribute Group Definitions

3.6.1 The Attribute Group Definition Schema Component

Attribute group definitions do not participate in validation as such, but the {attribute uses} and {attribute wildcard} of one or more complex type definitions may be constructed in whole or part by reference to an attribute group. Thus, attribute group definitions provide a replacement for some uses of XML’s parameter entity facility. Attribute group definitions are provided primarily for reference from the XML representation of schema components (see <complexType> and <attributeGroup>).

Example

```xml
<xs:attributeGroup name="myAttrGroup">
   <xs:attribute . . ./>
   . . .
</xs:attributeGroup>

<xs:complexType name="myelement">
   . . .
   <xs:attributeGroup ref="myAttrGroup"/>
</xs:complexType>
```

XML representations for attribute group definitions. The effect is as if the attribute declarations in the group were present in the type definition.

3.6.1 The Attribute Group Definition Schema Component

The attribute group definition schema component has the following properties:

Schema Component: Attribute Group Definition

{name}
An NCName as defined by [XML−Namespaces].
{target namespace}
Either absent or a namespace name, as defined in [XML−Namespaces].
{attribute uses}
A set of attribute uses.
{attribute wildcard}
Optional. A wildcard.
{annotation}
Optional. An annotation.

Attribute groups are identified by their {name} and {target namespace}; attribute group identities must be unique within an XML Schema. See References to schema components across namespaces (§4.2.3) for the use of component identifiers when importing one schema into another.
{attribute uses} is a set attribute uses, allowing for local specification of occurrence and default or fixed values.

{attribute wildcard} provides for an attribute wildcard to be included in an attribute group. See above under Complex Type Definitions (§3.4) for the interpretation of attribute wildcards during validation.

See Annotations (§3.13) for information on the role of the {annotation} property.

3.6.2 XML Representation of Attribute Group Definition Schema Components

The XML representation for an attribute group definition schema component is an <attributeGroup> element information item. It provides for naming a group of attribute declarations and an attribute wildcard for use by reference in the XML representation of complex type definitions and other attribute group definitions. The correspondences between the properties of the information item and properties of the component it corresponds to are as follows:

XML Representation Summary: attributeGroup Element Information Item

```xml
<attributeGroup
   id = ID
   name = NCName
   ref = QName
   {any attributes with non−schema namespace . . .}>
   Content:  (annotation?, ((attribute | attributeGroup)*, anyAttribute?))
</attributeGroup>
```

When an <attributeGroup> appears as a daughter of <schema> or <redefine>, it corresponds to an attribute group definition as below. When it appears as a daughter of <complexType> or <attributeGroup>, it does not correspond to any component as such.

**Attribute Group Definition Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{name}</td>
<td>The actual value of the name [attribute]</td>
</tr>
<tr>
<td>{target namespace}</td>
<td>The actual value of the targetNamespace [attribute] of the parent schema element information item.</td>
</tr>
<tr>
<td>{attribute uses}</td>
<td>The union of the set of attribute uses corresponding to the &lt;attribute&gt; [children], if any, with the {attribute uses} of the attribute groups -resolved- to by the -actual value-s of the ref [attribute] of the &lt;attributeGroup&gt; [children], if any.</td>
</tr>
<tr>
<td>{attribute wildcard}</td>
<td>As for the -complete wildcard- as described in XML Representation of Complex Type Definitions (§3.4.2).</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotation corresponding to the &lt;annotation&gt; element information item in the [children], if present, otherwise -absent-.</td>
</tr>
</tbody>
</table>

The example above illustrates a pattern which recurs in the XML representation of schemas: The same element, in this case attributeGroup, serves both to define and to incorporate by reference. In the first case the name attribute is required, in the second the ref attribute is required, and the element must be empty. These two are mutually exclusive, and also conditioned by context: the defining form, with a name,
must occur at the top level of a schema, whereas the referring form, with a ref, must occur within a complex type definition or an attribute group definition.

3.6.3 Constraints on XML Representations of Attribute Group Definitions

Schema Representation Constraint: Attribute Group Definition Representation OK
In addition to the conditions imposed on <attributeGroup> element information items by the schema for schemas, all of the following must be true:1 The corresponding attribute group definition, if any, must satisfy the conditions set out in Constraints on Attribute Group Definition Schema Components (§3.6.6) 2 If clause 2.1 or clause 2.2 in the correspondence specification in XML Representation of Complex Type Definitions (§3.4.2) for [attribute wildcard], as referenced above, is satisfied, the intensional intersection must be expressible, as defined in Attribute Wildcard Intersection (§3.10.6). 3 Circular group reference is disallowed outside <redefine>. That is, unless this element information item's parent is <redefine>, then among the [children], if any, there must not be an <attributeGroup> with ref [attribute] which resolves to the component corresponding to this <attributeGroup>.

3.6.4 Attribute Group Definition Validation Rules
None as such.

3.6.5 Attribute Group Definition Information Set Contributions
None as such.

3.6.6 Constraints on Attribute Group Definition Schema Components
All attribute group definitions (see Attribute Group Definitions (§3.6)) must satisfy the following constraint.

Schema Component Constraint: Attribute Group Definition Properties Correct
All of the following must be true:1 The values of the properties of an attribute group definition must be as described in the property tableau in The Attribute Group Definition Schema Component (§3.6.1), modulo the impact of Missing Sub−components (§5.3); 2 Two distinct members of the {attribute uses} must not have {attribute declaration}s both of whose {name}s match and whose {target namespace}s are identical. 3 Two distinct members of the {attribute uses} must not have {attribute declaration}s both of whose {type definition}s are or are derived from ID.

3.7 Model Group Definitions

3.7.1 The Model Group Definition Schema Component
3.7.2 XML Representation of Model Group Definition Schema Components
3.7.3 Constraints on XML Representations of Model Group Definitions
3.7.4 Model Group Definition Validation Rules
3.7.5 Model Group Definition Information Set Contributions
3.7.6 Constraints on Model Group Definition Schema Components

A model group definition associates a name and optional annotations with a Model Group (§2.2.3.1). By reference to the name, the entire model group can be incorporated by reference into a {term}.

Model group definitions are provided primarily for reference from the XML Representation of Complex Type Definitions (§3.4.2) (see <complexType> and <group>). Thus, model group definitions provide a replacement for some uses of XML’s parameter entity facility.

Example
A minimal model group is defined and used by reference, first as the whole content model, then as one alternative in a choice.

### 3.7.1 The Model Group Definition Schema Component

The model group definition schema component has the following properties:

- **Schema Component:** Model Group Definition
  - **{name}**
    - An NCName as defined by [XML−Namespaces].
  - **{target namespace}**
    - Either ·absent· or a namespace name, as defined in [XML−Namespaces].
  - **{model group}**
    - A model group.
  - **{annotation}**
    - Optional. An annotation.

Model group definitions are identified by their `{name}` and `{target namespace}`: model group identities must be unique within an -XML Schema-. See References to schema components across namespaces (§4.2.3) for the use of component identifiers when importing one schema into another.

Model group definitions *per se* do not participate in _validation_, but the `{term}` of a particle may correspond in whole or in part to a model group from a model group definition.

- **{model group}** is the Model Group (§2.2.3.1) for which the model group definition provides a name.

See Annotations (§3.13) for information on the role of the `{annotation}` property.

### 3.7.2 XML Representation of Model Group Definition Schema Components

The XML representation for a model group definition schema component is a `<group>` element information item. It provides for naming a model group for use by reference in the XML representation of complex type definitions and model groups. The correspondences between the properties of the information item and
properties of the component it corresponds to are as follows:

XML Representation Summary: group Element Information Item

```xml
<group
    name = NCName>
    Content:  (annotation?, (all | choice | sequence))
</group>
```

If there is a `name [attribute]` (in which case the item will have `<schema>` or `<redefine>` as parent), then the item corresponds to a model group definition component with properties as follows:

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{name}</code></td>
<td>The ·actual value· of the <code>name [attribute]</code></td>
</tr>
<tr>
<td><code>{target namespace}</code></td>
<td>The ·actual value· of the <code>targetNamespace [attribute]</code> of the parent schema element information item.</td>
</tr>
<tr>
<td><code>{model group}</code></td>
<td>A model group which is the <code>{term}</code> of a particle corresponding to the <code>&lt;all&gt;</code>, <code>&lt;choice&gt;</code> or <code>&lt;sequence&gt;</code> among the <code>{children}</code> (there must be one).</td>
</tr>
<tr>
<td><code>{annotation}</code></td>
<td>The annotation corresponding to the <code>&lt;annotation&gt;</code> element information item in the <code>{children}</code>, if present, otherwise ·absent·.</td>
</tr>
</tbody>
</table>

Otherwise, the item will have a `ref [attribute]`, in which case it corresponds to a particle component with properties as follows (unless `minOccurs=maxOccurs=0`, in which case the item corresponds to no component at all):

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{min occurs}</code></td>
<td>The ·actual value· of the <code>minOccurs [attribute]</code>, if present, otherwise 1.</td>
</tr>
<tr>
<td><code>{max occurs}</code></td>
<td><code>unbounded</code>, if the <code>maxOccurs [attribute]</code> equals <code>unbounded</code>, otherwise the ·actual value· of the <code>maxOccurs [attribute]</code>, if present, otherwise 1.</td>
</tr>
<tr>
<td><code>{term}</code></td>
<td>The <code>{model group}</code> of the model group definition ·resolved·, to by the ·actual value· of the <code>ref [attribute]</code>.</td>
</tr>
</tbody>
</table>

The name of this section is slightly misleading, in that the second, un−named, case above (with a `ref` and no `name`) is not really a named model group at all, but a reference to one. Also note that in the first (named) case above no reference is made to `minOccurs` or `maxOccurs`: this is because the schema for schemas does not allow them on the child of `<group>` when it is named. This in turn is because the `{min occurs}` and `{max occurs}` of the particles which refer to the definition are what count.

Given the constraints on its appearance in content models, an `<all>` should only occur as the only item in the `{children}` of a named model group definition or a content model: see Constraints on Model Group Schema Components (§3.8.6).
3.7.3 Constraints on XML Representations of Model Group Definitions

Schema Representation Constraint: Model Group Definition Representation OK
In addition to the conditions imposed on <group> element information items by the schema for schemas, the corresponding model group definition, if any, must satisfy the conditions set out in Constraints on Model Group Schema Components (§3.8.6).

3.7.4 Model Group Definition Validation Rules

None as such.

3.7.5 Model Group Definition Information Set Contributions

None as such.

3.7.6 Constraints on Model Group Definition Schema Components

All model group definitions (see Model Group Definitions (§3.7)) must satisfy the following constraint.

Schema Component Constraint: Model Group Definition Properties Correct
The values of the properties of a model group definition must be as described in the property tableau in The Model Group Definition Schema Component (§3.7.1), modulo the impact of Missing Sub–components (§5.3).

3.8 Model Groups

3.8.1 The Model Group Schema Component
3.8.2 XML Representation of Model Group Schema Components
3.8.3 Constraints on XML Representations of Model Groups
3.8.4 Model Group Validation Rules
3.8.5 Model Group Information Set Contributions
3.8.6 Constraints on Model Group Schema Components

When the [children] of element information items are not constrained to be empty or by reference to a simple type definition (Simple Type Definitions (§3.14)), the sequence of element information item [children] content may be specified in more detail with a model group. Because the [term] property of a particle can be a model group, and model groups contain particles, model groups can indirectly contain other model groups; the grammar for content models is therefore recursive.

Example

```xml
<xs:all>
  <xs:element ref="cats"/>
  <xs:element ref="dogs"/>
</xs:all>

<xs:sequence>
  <xs:choice>
    <xs:element ref="left"/>
    <xs:element ref="right"/>
  </xs:choice>
  <xs:element ref="landmark"/>
</xs:sequence>
```

XML representations for the three kinds of model group, the third nested inside the second.
3.8.1 The Model Group Schema Component

The model group schema component has the following properties:

Schema Component: Model Group

{compositor}
    One of all, choice or sequence.
{particles}
    A list of particles
{annotation}
    Optional. An annotation.

specifies a sequential (sequence), disjunctive (choice) or conjunctive (all) interpretation of the {particles}. This in turn determines whether the element information item [children] validated by the model group must:

- (sequence) correspond, in order, to the specified {particles};
- (choice) corresponded to exactly one of the specified {particles};
- (all) contain all and only exactly zero or one of each element specified in {particles}. The elements can occur in any order. In this case, to reduce implementation complexity, {particles} is restricted to contain local and top-level element declarations only, with {min occurs} = 0 or 1, {max occurs} = 1.

When two or more particles contained directly or indirectly in the {particles} of a model group have identically named element declarations as their {term}, the type definitions of those declarations must be the same. By 'indirectly' is meant particles within the {particles} of a group which is itself the {term} of a directly contained particle, and so on recursively.

See Annotations (§3.13) for information on the role of the {annotation} property.

3.8.2 XML Representation of Model Group Schema Components

The XML representation for a model group schema component is either an <all>, a <choice> or a <sequence> element information item. The correspondences between the properties of those information items and properties of the component they correspond to are as follows:

XML Representation Summary: all Element Information Item

<all
    id = ID
    maxOccurs = 1
    minOccurs = (0 | 1): 1
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, element*)
</all>

<choice
    id = ID
    maxOccurs = (nonNegativeInteger | unbounded) : 1
    minOccurs = nonNegativeInteger : 1
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, (element | group | choice | sequence | any)*)

3.8 Model Groups
Each of the above items corresponds to a particle containing a model group, with properties as follows (unless minOccurs=maxOccurs=0, in which case the item corresponds to no component at all):

### Particle Schema Component

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>minOccurs</td>
<td>The actual value of the minOccurs attribute, if present, otherwise 1.</td>
</tr>
<tr>
<td>maxOccurs</td>
<td>unbounded, if the maxOccurs attribute equals unbounded, otherwise the actual value of the maxOccurs attribute, if present, otherwise 1.</td>
</tr>
</tbody>
</table>

### Model Group Schema Component

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>compositor</td>
<td>One of all, choice, sequence depending on the element information item.</td>
</tr>
<tr>
<td>particles</td>
<td>A sequence of particles corresponding to all the &lt;all&gt;, &lt;choice&gt;, &lt;sequence&gt;, &lt;any&gt;, &lt;group&gt; or &lt;element&gt; items among the [children], in order.</td>
</tr>
<tr>
<td>annotation</td>
<td>The annotation corresponding to the &lt;annotation&gt; element information item in the [children], if present, otherwise absent.</td>
</tr>
</tbody>
</table>

#### 3.8.3 Constraints on XML Representations of Model Groups

**Schema Representation Constraint: Model Group Representation OK**

In addition to the conditions imposed on <all>, <choice> and <sequence> element information items by the schema for schemas, the corresponding particle and model group must satisfy the conditions set out in Constraints on Model Group Schema Components (§3.8.6) and Constraints on Particle Schema Components (§3.9.6).

#### 3.8.4 Model Group Validation Rules

**Validation Rule: Element Sequence Valid**

[Definition:] Define a partition of a sequence as a sequence of sub−sequences, some or all of which may be empty, such that concatenating all the sub−sequences yields the original sequence.

For a sequence (possibly empty) of element information items to be locally valid, with respect to a model group the appropriate case among the following must be true: 1 If the compositor is sequence, then there must be a partition of the sequence into n sub−sequences where n is the length of particles such that each of the sub−sequences in order is valid, with respect to the corresponding particle in the particles as defined...
2 If the {compositor} is choice, then there must be a particle among the {particles} such that the sequence is ·valid· with respect to that particle as defined in Element Sequence Locally Valid (Particle) (§3.9.4).

3 If the {compositor} is all, then there must be a ·partition· of the sequence into \( n \) sub-sequences where \( n \) is the length of {particles} such that there is a one-to-one mapping between the sub-sequences and the {particles} where each sub-sequence is ·valid· with respect to the corresponding particle as defined in Element Sequence Locally Valid (Particle) (§3.9.4).

Nothing in the above should be understood as ruling out groups whose {particles} is empty: although no sequence can be ·valid· with respect to such a group whose {compositor} is choice, the empty sequence is ·valid· with respect to empty groups whose {compositor} is sequence or all.

NOTE: The above definition is implicitly non-deterministic, and should not be taken as a recipé for implementations. Note in particular that when {compositor} is all, particles is restricted to a list of local and top-level element declarations (see Constraints on Model Group Schema Components (§3.8.6)). A much simpler implementation is possible than would arise from a literal interpretation of the definition above; informally, the content is ·valid· when each declared element occurs exactly once (or at most once, if \( \text{min occurs} = 0 \)), and each is ·valid· with respect to its corresponding declaration. The elements can occur in arbitrary order.

### 3.8.5 Model Group Information Set Contributions

None as such.

### 3.8.6 Constraints on Model Group Schema Components

All model groups (see Model Groups (§3.8)) must satisfy the following constraints.

**Schema Component Constraint: Model Group Correct**

All of the following must be true:
1. The values of the properties of a model group must be as described in the property tableau in The Model Group Schema Component (§3.8.1), modulo the impact of Missing Sub-Components (§5.3).
2. Circular groups are disallowed. That is, within the {particles} of a group there must not be at any depth a particle whose {term} is the group itself.

**Schema Component Constraint: All Group Limited**

When a model group has {compositor} all of the following must be true:
1. All of the following must be true:
   1.1 It appears as the model group of a model group definition.
   1.2 It appears in a particle with \( \text{min occurs} = \text{max occurs} = 1 \), and that particle must be part of a pair which constitutes the {content type} of a complex type definition.
2. The \( \text{max occurs} \) of all the particles in the {particles} of the group must be 0 or 1.

**Schema Component Constraint: Element Declarations Consistent**

If the {particles} contains, either directly, indirectly (that is, within the {particles} of a contained model group, recursively) or implicitly, two or more element declaration particles with the same {name} and {target namespace}, then all their type definitions must be the same top-level definition, that is, all of the following must be true:
1. All their {type definition} must have a non-absent name.
2. All their {type definition} must have the same name.
3. All their {type definition} must have the same target namespace.

[Definition:] A list of particles implicitly contains an element declaration if a member of the list contains that element declaration in its -substitution group.

**Schema Component Constraint: Unique Particle Attribution**

A content model must be formed such that during -validation of an element information item sequence, the particle contained directly, indirectly or implicitly therein with which to attempt to -validate each item in the sequence in turn can be uniquely determined without examining the content or attributes of that item, and...
NOTE: This constraint reconstructs for XML Schema the equivalent constraints of [XML 1.0 (Second Edition)] and SGML. Given the presence of element substitution groups and wildcards, the concise expression of this constraint is difficult, see Analysis of the Unique Particle Attribution Constraint (non−normative) (§H) for further discussion.

NOTE: Because locally−scoped element declarations may or may not have a [target namespace], the scope of declarations is not relevant to enforcing either of the two preceding constraints.

The following constraints define relations appealed to elsewhere in this specification.

Schema Component Constraint: Effective Total Range (all and sequence)
The effective total range of a particle whose {term} is a group whose {compositor} is all or sequence is a pair of minimum and maximum, as follows:

minimum
The product of the particle's {min occurs} and the sum of the {min occurs} of every wildcard or element declaration particle in the group's {particles} and the minimum part of the effective total range of each of the group particles in the group's {particles} (or 0 if there are no {particles}).

maximum
unbounded if the {max occurs} of any wildcard or element declaration particle in the group's {particles} or the maximum part of the effective total range of any of the group particles in the group's {particles} is unbounded, or if any of those is non−zero and the {max occurs} of the particle itself is unbounded, otherwise the product of the particle's {max occurs} and the sum of the {max occurs} of every wildcard or element declaration particle in the group's {particles} and the maximum part of the effective total range of each of the group particles in the group's {particles} (or 0 if there are no {particles}).

Schema Component Constraint: Effective Total Range (choice)
The effective total range of a particle whose {term} is a group whose {compositor} is choice is a pair of minimum and maximum, as follows:

minimum
The product of the particle's {min occurs} and the minimum of the {min occurs} of every wildcard or element declaration particle in the group's {particles} and the minimum part of the effective total range of each of the group particles in the group's {particles} (or 0 if there are no {particles}).

maximum
unbounded if the {max occurs} of any wildcard or element declaration particle in the group's {particles} or the maximum part of the effective total range of any of the group particles in the group's {particles} is unbounded, or if any of those is non−zero and the {max occurs} of the particle itself is unbounded, otherwise the product of the particle's {max occurs} and the maximum of the {max occurs} of every wildcard or element declaration particle in the group's {particles} and the maximum part of the effective total range of each of the group particles in the group's {particles} (or 0 if there are no {particles}).

3.9 Particles

3.9.1 The Particle Schema Component3.9.2 XML Representation of Particle Components3.9.3 Constraints on XML Representations of Particles3.9.4 Particle Validation Rules3.9.5 Particle Information Set
Contributions

3.9.6 Constraints on Particle Schema Components

As described in Model Groups (§3.8), particles contribute to the definition of content models.

Example

```xml
<xs:element ref="egg" minOccurs="12" maxOccurs="12"/>
<xs:group ref="omelette" minOccurs="0"/>
<xs:any maxOccurs="unbounded"/>
```

XML representations which all involve particles, illustrating some of the possibilities for controlling occurrence.

3.9.1 The Particle Schema Component

The particle schema component has the following properties:

Schema Component: Particle

- `{min occurs}`
  - A non-negative integer.
- `{max occurs}`
  - Either a non-negative integer or unbounded.
- `{term}`
  - One of a model group, a wildcard, or an element declaration.

In general, multiple element information item [children], possibly with intervening character [children] if the content type is mixed, can be validated with respect to a single particle. When the `{term}` is an element declaration or wildcard, `{min occurs}` determines the minimum number of such element [children] that can occur. The number of such children must be greater than or equal to `{min occurs}`. If `{min occurs}` is 0, then occurrence of such children is optional.

Again, when the `{term}` is an element declaration or wildcard, the number of such element [children] must be less than or equal to any numeric specification of `{max occurs}`; if `{max occurs}` is unbounded, then there is no upper bound on the number of such children.

When the `{term}` is a model group, the permitted occurrence range is determined by a combination of `{min occurs}` and `{max occurs}` and the occurrence ranges of the `{term}`’s `{particles}`.

3.9.2 XML Representation of Particle Components

Particles correspond to all three elements `<element>` not immediately within `<schema>`, `<group>` not immediately within `<schema>` and `<any>` which allow minOccurs and maxOccurs attributes. These in turn correspond to two components in each case, a particle and its `{term}`. The appropriate mapping is described in XML Representation of Element Declaration Schema Components (§3.3.2), XML Representation of Model Group Schema Components (§3.8.2) and XML Representation of Wildcard Schema Components (§3.10.2) respectively.
3.9.3 Constraints on XML Representations of Particles

None as such.

3.9.4 Particle Validation Rules

Validation Rule: Element Sequence Locally Valid (Particle)
For a sequence (possibly empty) of element information items to be locally valid with respect to a particle the appropriate case among the following must be true: 1. If the term is a wildcard, then all of the following must be true: 1.1 The length of the sequence must be greater than or equal to the min occurs. 1.2 If max occurs is a number, the length of the sequence must be less than or equal to the max occurs. 1.3 Each element information item in the sequence must be valid with respect to the wildcard as defined by Item Valid (Wildcard) (§3.10.4). 2 If the term is an element declaration, then all of the following must be true: 2.1 The length of the sequence must be greater than or equal to the min occurs. 2.2 If max occurs is a number, the length of the sequence must be less than or equal to the max occurs. 2.3 For each element information item in the sequence one of the following must be true: 2.3.1 The element declaration is local (i.e. its scope must not be global), its abstract is false, the element information item's namespace name is identical to the element declaration's target namespace (where an absent target namespace is taken to be identical to a namespace name with no value) and the element information item's local name matches the element declaration's name.

In this case the element declaration is the context–determined declaration for the element information item with respect to Schema–Validity Assessment (Element) (§3.3.4) and Assessment Outcome (Element) (§3.3.5). 2.3.2 The element declaration is top–level (i.e. its scope is global), its abstract is false, the element information item's namespace name is identical to the element declaration's target namespace (where an absent target namespace is taken to be identical to a namespace name with no value) and the element information item's local name matches the element declaration's name.

In this case the element declaration is the context–determined declaration for the element information item with respect to Schema–Validity Assessment (Element) (§3.3.4) and Assessment Outcome (Element) (§3.3.5). 2.3.3 The element declaration is top–level (i.e. its scope is global), its disallowed substitutions does not contain substitution, the local and namespace name of the element information item resolve to an element declaration, as defined in QName resolution (Instance) (§3.15.4). -- [Definition:] call this declaration the substituting declaration and the substituting declaration together with the particle's element declaration's disallowed substitutions is validly substitutable for the particle's element declaration as defined in Substitution Group OK (Transitive) (§3.3.6).

In this case the substituting declaration is the context–determined declaration for the element information item with respect to Schema–Validity Assessment (Element) (§3.3.4) and Assessment Outcome (Element) (§3.3.5). 3 If the term is a model group, then all of the following must be true: 3.1 There is a partition of the sequence into n sub–sequences such that n is greater than or equal to min occurs. 3.2 If max occurs is a number, n must be less than or equal to max occurs. 3.3 Each sub–sequence in the partition is valid with respect to that model group as defined in Element Sequence Valid (§3.8.4).

NOTE: Clauses clause 1 and clause 2.3.3 do not interact: an element information item validatable by a declaration with a substitution group head in a different namespace is not validatable by a wildcard which accepts the head's namespace but not its own.
3.9.5 Particle Information Set Contributions

None as such.

3.9.6 Constraints on Particle Schema Components

All particles (see Particles (§3.9)) must satisfy the following constraints.

Schema Component Constraint: Particle Correct

All of the following must be true: 1 The values of the properties of a particle must be as described in the property tableau in The Particle Schema Component (§3.9.1), modulo the impact of Missing Sub−components (§5.3). 2 If \( \text{max occurs} \) is not unbounded, that is, it has a numeric value, then all of the following must be true: 2.1 \( \text{min occurs} \) must not be greater than \( \text{max occurs} \). 2.2 \( \text{max occurs} \) must be greater than or equal to 1.

The following constraints define relations appealed to elsewhere in this specification.

Schema Component Constraint: Particle Valid (Extension)

[Definition:] For a particle (call it \( E \), for extension) to be a valid extension of another particle (call it \( B \), for base) one of the following must be true: 1 They are the same particle. 2 \( E \)'s \( \{ \text{min occurs} \} = \{ \text{max occurs} \} = 1 \) and its \( \{ \text{term} \} \) is a sequence group whose \( \{ \text{particles} \} \) first member is a particle all of whose properties, recursively, are identical to those of \( B \), with the exception of \( \{ \text{annotation} \} \) properties.

The approach to defining a type by restricting another type definition set out here is designed to ensure that types defined in this way are guaranteed to be a subset of the type they restrict. This is accomplished by requiring a clear mapping between the components of the base type definition and the restricting type definition. Permissible mappings are set out below via a set of recursive definitions, bottoming out in the obvious cases, e.g. where an (restricted) element declaration corresponds to another (base) element declaration with the same name and type but the same or wider range of occurrence.

NOTE: The structural correspondence approach to guaranteeing the subset relation set out here is necessarily verbose, but has the advantage of being checkable in a straightforward way. The working group solicits feedback on how difficult this is in practice, and on whether other approaches are found to be viable.

Schema Component Constraint: Particle Valid (Restriction)

[Definition:] For a particle (call it \( R \), for restriction) to be a valid restriction of another particle (call it \( B \), for base) one of the following must be true: 1 They are the same particle. 2 depending on the kind of particle, per the table below, with the qualifications that all of the following must be true: 2.1 Any top−level element declaration particle (in \( R \) or \( B \)) which is the \{ substitution group affiliation \} of one or more other element declarations is treated as if it were a choice group whose \( \{ \text{particles} \} \) first member is a particle all of whose properties, recursively, are identical to those of \( B \), with the exception of \( \{ \text{annotation} \} \) properties. 2.2 Any pointless occurrences of \(<\text{sequence}>\), \(<\text{choice}>\) or \(<\text{all}>\) are ignored, where pointlessness is understood as follows:

\(<\text{sequence}>\)

One of the following must be true: 2.2.1 \( \{ \text{particles} \} \) is empty. 2.2.2 All of the following must be true: 2.2.2.1 The particle within which this \(<\text{sequence}>\) appears has \{ \text{max occurs} \} and \{ \text{min occurs} \} of 1. 2.2.2.2 One of the following must be true: 2.2.2.2.1 The \(<\text{sequence}>\)'s \( \{ \text{particles} \} \) has only one member. 2.2.2.2.2 The particle within which this \(<\text{sequence}>\) appears is itself among the \( \{ \text{particles} \} \) of a \(<\text{sequence}>\).
<all>
One of the following must be true: 2.2.1 {particles} is empty. 2.2.2 {particles} has only one member.
</all>

<choice>
One of the following must be true: 2.2.1 {particles} is empty and the particle within which this <choice> appears has \{min occurs\} of 0. 2.2.2 All of the following must be true: 2.2.2.1 The particle within which this <choice> appears has \{max occurs\} and \{min occurs\} of 1. 2.2.2.2 One of the following must be true: 2.2.2.2.1 The <choice>'s {particles} has only one member. 2.2.2.2.2 The particle within which this <choice> appears is itself among the {particles} of a <choice>.
</choice>

### Base Particle

<table>
<thead>
<tr>
<th></th>
<th>elt</th>
<th>any</th>
<th>all</th>
<th>choice</th>
<th>sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derived Particle</td>
<td>NameAndTypeOK</td>
<td>NSCompat</td>
<td>Recurse--</td>
<td>Recurse--</td>
<td>RecurseAs--</td>
</tr>
<tr>
<td>any</td>
<td>Forbidden</td>
<td>NSSubset</td>
<td>Forbidden</td>
<td>Forbidden</td>
<td>Forbidden</td>
</tr>
<tr>
<td>all</td>
<td>Forbidden</td>
<td>NSRecurse--CheckCardinality</td>
<td>Recurse</td>
<td>Forbidden</td>
<td>Forbidden</td>
</tr>
<tr>
<td>choice</td>
<td>Forbidden</td>
<td>NSRecurse--CheckCardinality</td>
<td>Forbidden</td>
<td>RecurseLax</td>
<td>Forbidden</td>
</tr>
<tr>
<td>sequence</td>
<td>Forbidden</td>
<td>NSRecurse--CheckCardinality</td>
<td>Recurse--</td>
<td>Unordered</td>
<td>MapAndSum</td>
</tr>
</tbody>
</table>

#### Schema Component Constraint: Occurrence Range OK
For a particle's occurrence range to be a valid restriction of another's occurrence range all of the following must be true: 1. Its \{min occurs\} is greater than or equal to the other's \{min occurs\}. 2. The other's \{max occurs\} is unbounded. 2.2 Both \{max occurs\} are numbers, and the particle's is less than or equal to the other's. **Schema Component Constraint: Particle Restriction OK**

(Elt:Elt -- NameAndTypeOK)
For an element declaration particle to be a valid restriction of another element declaration particle all of the following must be true: 1. The declarations' \{name\} and \{target namespace\} are the same. 2. Either B's \{nillable\} is true or R's \{nillable\} is false. 3. R's occurrence range is a valid restriction of B's occurrence range as defined by Occurrence Range OK (§3.9.6). 4. Either B's declaration's \{value constraint\} is absent, or is not fixed, or R's declaration's \{value constraint\} is fixed with the same value. 5. R's declaration's \{identity-constraint definitions\} is a subset of B's declaration's \{identity-constraint definitions\}, if any. 6. R's declaration's \{disallowed substitutions\} is a superset of B's declaration's \{disallowed substitutions\}. 7. R's \{type definition\} is validly derived given \{extension, list, union\} from B's \{type definition\} as defined by Type Derivation OK (Complex) (§3.4.6) or Type Derivation OK (Simple) (§3.14.6), as appropriate.

**NOTE:** The above constraint on \{type definition\} means that in deriving a type by restriction, any contained type definitions must themselves be explicitly derived by restriction from the corresponding type definitions in the base definition.

#### Schema Component Constraint: Particle Derivation OK (Elt:Any -- NSCompat)
For an element declaration particle to be a valid restriction of a wildcard particle all of the following must be true: 1. The element declaration's \{target namespace\} is valid with respect to the wildcard's \{namespace constraint\} as defined by Wildcard allows Namespace Name (§3.10.4). 2. R's occurrence range is a valid restriction of B's occurrence range as defined by Occurrence Range OK (§3.9.6). **Schema Component**
Constraint: Particle Derivation OK (Elt:All/Choice/Sequence −− RecurseAsIfGroup)
For an element declaration particle to be a ·valid restriction· of a group particle (all, choice or sequence) a group particle of the variety corresponding to B's, with [min occurs] and [max occurs] of 1 and with {particles} consisting of a single particle the same as the element declaration must be a ·valid restriction· of the group as defined by Particle Derivation OK (All:All,Sequence:Sequence −− Recurse) (§3.9.6), Particle Derivation OK (Choice:Choice −− RecurseLax) (§3.9.6) or Particle Derivation OK (All:All,Sequence:Sequence −− Recurse) (§3.9.6), depending on whether the group is all, choice or sequence.

Schema Component Constraint: Particle Derivation OK (Any:Any −− NSSubset)
For a wildcard particle to be a ·valid restriction· of another wildcard particle all of the following must be true:1 R's occurrence range must be a valid restriction of B's occurrence range as defined by Occurrence Range OK (§3.9.6).2 R's {namespace constraint} must be an intensional subset of B's {namespace constraint} as defined by Wildcard Subset (§3.10.6).

Schema Component Constraint: Particle Derivation OK (All/Choice/Sequence:Any −− NSRecurseCheckCardinality)
For a group particle to be a ·valid restriction· of a wildcard particle all of the following must be true:1 Every member of the {particles} of the group is a ·valid restriction· of the wildcard as defined by Particle Valid (Restriction) (§3.9.6).2 The effective total range of the group, as defined by Effective Total Range (all and sequence) (§3.8.6) (if the group is all or sequence) or Effective Total Range (choice) (§3.8.6) (if it is choice) is a valid restriction of B's occurrence range as defined by Occurrence Range OK (§3.9.6).

NOTE: Although the -validation- semantics of an all group does not depend on the order of its particles, derived all groups are required to match the order of their base in order to simplify checking that the derivation is OK.

[Definition:] A complete functional mapping is order-preserving if each particle r in the domain R maps to a particle b in the range B which follows (not necessarily immediately) the particle in the range B mapped to by the predecessor of r, if any, where "predecessor" and "follows" are defined with respect to the order of the lists which constitute R and B.

Schema Component Constraint: Particle Derivation OK (Choice:Choice −− RecurseLax)
For a choice group particle to be a ·valid restriction· of another choice group particle all of the following must be true:1 R's occurrence range is a valid restriction of B's occurrence range as defined by Occurrence Range OK (§3.9.6).2 There is a complete order-preserving, functional mapping from the particles in the {particles} of R to the particles in the {particles} of B such that all of the following must be true:2.1 Each particle in the {particles} of R is a ·valid restriction· of the particle in the {particles} of B it maps to as defined by Particle Valid (Restriction) (§3.9.6).2.2 All particles in the {particles} of B which are not mapped to by any particle in the {particles} of R are ·emptiable· as defined by Particle Emptiable (§3.9.6).

NOTE: Although the -validation- semantics of a choice group does not depend on the order of its particles, derived choice groups are required to match the order of their base in order to simplify checking that the derivation is OK.

Schema Component Constraint: Particle Derivation OK (Sequence:All −− RecurseUnordered)
For a sequence group particle to be a ·valid restriction· of an all group particle all of the following must be true:1 R's occurrence range is a valid restriction of B's occurrence range as defined by Occurrence Range OK (§3.9.6).2 There is a complete functional mapping from the particles in the {particles} of R to the particles in...
the \{particles\} of B such that all of the following must be true:

1. No particle in the \{particles\} of B is mapped to by more than one of the particles in the \{particles\} of R;
2. Each particle in the \{particles\} of R is a valid restriction of the particle in the \{particles\} of B it maps to as defined by Particle Valid (Restriction) (§3.9.6);
3. All particles in the \{particles\} of B which are not mapped to by any particle in the \{particles\} of R are emptiable as defined by Particle Emptiable (§3.9.6).

NOTE: Although this clause allows reordering, because of the limits on the contents of all groups the checking process can still be deterministic.

Schema Component Constraint: Particle Derivation OK (Sequence:Choice -- MapAndSum)

For a sequence group particle to be a valid restriction of a choice group particle all of the following must be true:

1. There is a complete functional mapping from the particles in the \{particles\} of R to the particles in the \{particles\} of B such that each particle in the \{particles\} of R is a valid restriction of the particle in the \{particles\} of B it maps to as defined by Particle Valid (Restriction) (§3.9.6).
2. The pair consisting of the product of the \{min occurs\} of R and the length of its \{particles\} and \textit{unbounded} if \{max occurs\} is \textit{unbounded} otherwise the product of the \{max occurs\} of R and the length of its \{particles\} is a valid restriction of B's occurrence range as defined by Occurrence Range OK (§3.9.6).

NOTE: This clause is in principle more restrictive than absolutely necessary, but in practice will cover all the likely cases, and is much easier to specify than the fully general version.

NOTE: This case allows the "unfolding" of iterated disjunctions into sequences. It may be particularly useful when the disjunction is an implicit one arising from the use of substitution groups.

Schema Component Constraint: Particle Emptiable

[Definition:] For a particle to be \textit{emptiable} one of the following must be true:

1. Its \{min occurs\} is \textit{0}.
2. Its \{term\} is a group and the minimum part of the effective total range of that group, as defined by Effective Total Range (all and sequence) (§3.8.6) (if the group is \textit{all} or \textit{sequence}) or Effective Total Range (choice) (§3.8.6) (if it is \textit{choice}), is \textit{0}.

3.10 Wildcards

3.10.1 The Wildcard Schema Component

In order to exploit the full potential for extensibility offered by XML plus namespaces, more provision is needed than DTDs allow for targeted flexibility in content models and attribute declarations. A wildcard provides for validation of attribute and element information items dependent on their namespace name, but independently of their local name.

Example

```xml
<xs:any processContents="skip"/>
<xs:any namespace="##other" processContents="lax"/>
<xs:any namespace="http://www.w3.org/1999/XSL/Transform"/>
<xs:any namespace="##targetNamespace"/>
```
XML representations of the four basic types of wildcard, plus one attribute wildcard.

### 3.10.1 The Wildcard Schema Component

The wildcard schema component has the following properties:

**Schema Component:** Wildcard

**{namespace constraint}**

One of any; a pair of not and a namespace name or absent; or a set whose members are either namespace names or absent.

**{process contents}**

One of skip, lax or strict.

**{annotation}**

Optional. An annotation.

{namespace constraint} provides for validation of attribute and element items that:

1. (any) have any namespace or are not namespace qualified;
2. (not and a namespace name) have any namespace other than the specified namespace name, or are not namespace qualified;
3. (not and absent) are namespace qualified;
4. (a set whose members are either namespace names or absent) have any of the specified namespaces and/or, if absent is included in the set, are unqualified.

{process contents} controls the impact on assessment of the information items allowed by wildcards, as follows:

**strict**

There must be a top-level declaration for the item available, or the item must have an xsi:type, and the item must be valid, as appropriate.

**skip**

No constraints at all: the item must simply be well-formed XML.

**lax**

If the item, or any items among its children if it's an element information item, has a uniquely determined declaration available, it must be valid with respect to that definition, that is, validate where you can, don't worry when you can't.

See Annotations (§3.13) for information on the role of the {annotation} property.

### 3.10.2 XML Representation of Wildcard Schema Components

The XML representation for a wildcard schema component is an <any> or <anyAttribute> element information item. The correspondences between the properties of an <any> information item and properties of the components it corresponds to are as follows (see <complexType> and <attributeGroup> for the correspondences for <anyAttribute>):

**XML Representation Summary:** any Element Information Item
A particle containing a wildcard, with properties as follows (unless minOccurs=maxOccurs=0, in which case the item corresponds to no component at all):

**Particle Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{min occurs}</td>
<td>The <em>actual value</em> of the minOccurs [attribute], if present, otherwise 1.</td>
</tr>
<tr>
<td>{max occurs}</td>
<td>unbounded, if the maxOccurs [attribute] equals unbounded, otherwise the <em>actual value</em> of the maxOccurs [attribute], if present, otherwise 1.</td>
</tr>
<tr>
<td>{term}</td>
<td>A wildcard as given below:</td>
</tr>
</tbody>
</table>

**Wildcard Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{namespace constraint}</td>
<td>Dependent on the <em>actual value</em> of the namespace [attribute]: if absent, then any, otherwise as follows:</td>
</tr>
<tr>
<td></td>
<td>###any</td>
</tr>
<tr>
<td></td>
<td>any</td>
</tr>
<tr>
<td></td>
<td>###other</td>
</tr>
<tr>
<td></td>
<td>a pair of not and the <em>actual value</em> of the targetNamespace [attribute] of the &lt;schema&gt; ancestor element information item if present, otherwise absent.</td>
</tr>
<tr>
<td></td>
<td>otherwise</td>
</tr>
<tr>
<td></td>
<td>a set whose members are namespace names corresponding to the space–delimited substrings of the string, except if one such substring is ###targetNamespace, the corresponding member is the <em>actual value</em> of the targetNamespace [attribute] of the &lt;schema&gt; ancestor element information item if present, otherwise absent:2 if one such substring is ###local, the corresponding member is absent.</td>
</tr>
<tr>
<td>{process contents}</td>
<td>The <em>actual value</em> of the processContents [attribute], if present, otherwise strict.</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotation corresponding to the &lt;annotation&gt; element information item in the [children], if present, otherwise absent.</td>
</tr>
</tbody>
</table>

Wildcards are subject to the same ambiguity constraints (Unique Particle Attribution (§3.8.6)) as other content model particles: If an instance element could match either an explicit particle and a wildcard, or one of two wildcards, within the content model of a type, that model is in error.
3.10.3 Constraints on XML Representations of Wildcards

**Schema Representation Constraint: Wildcard Representation OK**

In addition to the conditions imposed on `<any>` element information items by the schema for schemas, the corresponding particle and model group must satisfy the conditions set out in Constraints on Model Group Schema Components (§3.8.6) and Constraints on Particle Schema Components (§3.9.6).

3.10.4 Wildcard Validation Rules

**Validation Rule: Item Valid (Wildcard)**

For an element or attribute information item to be locally ·valid· with respect to a wildcard constraint its [namespace name] must be ·valid· with respect to the wildcard constraint, as defined in Wildcard allows Namespace Name (§3.10.4).

When this constraint applies the appropriate case among the following must be true:1 If `{process contents}` is lax, then the item has no -context–determined declaration- with respect to Assessment Outcome (Element) (§3.3.5), Schema—Validity Assessment (Element) (§3.3.4) and Schema—Validity Assessment (Attribute) (§3.2.4).2 If `{process contents}` is strict, then the item's -context–determined declaration- is mustFind.3 If `{process contents}` is skip, then the item's -context–determined declaration- is skip.

**Validation Rule: Wildcard allows Namespace Name**

For a value which is either a namespace name or ·absent· to be ·valid· with respect to a wildcard constraint (the value of a [namespace constraint]) one of the following must be true:1 The constraint must be any.2 All of the following must be true:2.1 The constraint is a pair of not and a namespace name or ·absent·. ([Definition:] call this the namespace test ).2.2 The value must not be identical to the namespace test.2.3 The value must not be ·absent·.3 The constraint is a set, and the value is identical to one of the members of the set.

3.10.5 Wildcard Information Set Contributions

None as such.

3.10.6 Constraints on Wildcard Schema Components

All wildcards (see Wildcards (§3.10)) must satisfy the following constraint.

**Schema Component Constraint: Wildcard Properties Correct**

The values of the properties of a wildcard must be as described in the property tableau in The Wildcard Schema Component (§3.10.1), modulo the impact of Missing Sub–components (§5.3).

The following constraints define a relation appealed to elsewhere in this specification.

**Schema Component Constraint: Wildcard Subset**

For a namespace constraint (call it sub) to be an intensional subset of another namespace constraint (call it super) one of the following must be true:1 super must be any.2 All of the following must be true:2.1 sub must be a pair of not and a namespace name or ·absent·.2.2 super must be a pair of not and the same value.3 All of the following must be true:3.1 sub must be a set whose members are either namespace names or ·absent·.3.2 One of the following must be true:3.2.1 super must be the same set or a superset thereof.3.2.2 super must be a pair of not and a namespace name or ·absent· and that value must not be in sub's set.

**Schema Component Constraint: Attribute Wildcard Union**

For a wildcard's [namespace constraint] value to be the intensional union of two other such values (call them O1 and O2): the appropriate case among the following must be true:1 If O1 and O2 are the same value, then...
that value must be the value. If either $O_1$ or $O_2$ is any, then any must be the value. If both $O_1$ and $O_2$ are sets of (namespace names or \_\_absent\_), then the union of those sets must be the value. If the two are negations of different namespace names, then the intersection is not expressible. If either $O_1$ or $O_2$ is a pair of \textit{not} and a namespace name and the other is a set of (namespace names or \_\_absent\_), then the intersection is not expressible. If the two are negations of different namespace names, then the intersection is not expressible.

### 3.11 Identity–constraint Definitions

3.11.1 The Identity–constraint Definition Schema Component 3.11.2 XML Representation of Identity–constraint Definition Schema Components 3.11.3 Constraints on XML Representations of Identity–constraint Definitions 3.11.4 Identity–constraint Definition Validation Rules 3.11.5 Identity–constraint Definition Information Set Contributions 3.11.6 Constraints on Identity–constraint Definition Schema Components

Identity–constraint definition components provide for uniqueness and reference constraints with respect to the contents of multiple elements and attributes.

Example

```xml
<xs:key name="fullName">
  <xs:selector xpath="./person"/>
  <xs:field xpath="forename"/>
  <xs:field xpath="surname"/>
</xs:key>

<xs:keyref name="personRef" refer="fullName">
  <xs:selector xpath="./personPointer"/>
  <xs:field xpath="@first"/>
  <xs:field xpath="@last"/>
</xs:keyref>

<xs:unique name="nearlyID">
  <xs:selector xpath="/**"/>
  <xs:field xpath="@id"/>
</xs:unique>
```

XML representations for the three kinds of identity–constraint definitions.
3.11.1 The Identity–constraint Definition Schema Component

The identity–constraint definition schema component has the following properties:

Schema Component: Identity–constraint Definition

{name}
An NCName as defined by [XML–Namespaces].
	
{target namespace}
Either ·absent· or a namespace name, as defined in [XML–Namespaces].
	
{identity–constraint category}
One of key, keyref or unique.
	
{selector}
A restricted XPath ([XPath]) expression.
	
{fields}
A non–empty list of restricted XPath ([XPath]) expressions.
	
{referenced key}
Required if {identity–constraint category} is keyref, forbidden otherwise. An identity–constraint definition with {identity–constraint category} equal to key or unique.
	
{annotation}
Optional. An annotation.

Identity–constraint definitions are identified by their {name} and {target namespace}; Identity–constraint definition identities must be unique within an ·XML Schema·. See References to schema components across namespaces (§4.2.3) for the use of component identifiers when importing one schema into another.

Informally, {identity–constraint category} identifies the Identity–constraint definition as playing one of three roles:

• (unique) the Identity–constraint definition asserts uniqueness, with respect to the content identified by {selector}, of the tuples resulting from evaluation of the {fields} XPath expression(s).

• (key) the Identity–constraint definition asserts uniqueness as for unique. key further asserts that all selected content actually has such tuples.

• (keyref) the Identity–constraint definition asserts a correspondence, with respect to the content identified by {selector}, of the tuples resulting from evaluation of the {fields} XPath expression(s), with those of the {referenced key}.

These constraints are specified along side the specification of types for the attributes and elements involved, i.e. something declared as of type integer may also serve as a key. Each constraint declaration has a name, which exists in a single symbol space for constraints. The equality and inequality conditions appealed to in checking these constraints apply to the value of the fields selected, so that for example 3.0 and 3 would be conflicting keys if they were both number, but non–conflicting if they were both strings, or one was a string and one a number. Values of differing type can only be equal if one type is derived from the other, and the value is in the value space of both.

Overall the augmentations to XML’s ID/IDREF mechanism are:

• Functioning as a part of an identity–constraint is in addition to, not instead of, having a type;

• Not just attribute values, but also element content and combinations of values and content can be declared to be unique;

• Identity–constraints are specified to hold within the scope of particular elements;
(Combinations of) attribute values and/or element content can be declared to be keys, that is, not only unique, but always present and non-nillable;

• The comparison between `keyref {fields}` and `key or unique {fields}` is by value equality, not by string equality.

`{selector}` specifies a restricted XPath ([XPath]) expression relative to instances of the element being declared. This must identify a node set of subordinate elements (i.e. contained within the declared element) to which the constraint applies.

`{fields}` specifies XPath expressions relative to each element selected by a `selector`. This must identify a single node (element or attribute) whose content or value, which must be of a simple type, is used in the constraint. It is possible to specify an ordered list of `fields` to cater to multi-field keys, keyrefs, and uniqueness constraints.

In order to reduce the burden on implementers, in particular implementers of streaming processors, only restricted subsets of XPath expressions are allowed in `{selector}` and `{fields}`. The details are given in Constraints on Identity-constraint Definition Schema Components (§3.11.6).

**NOTE:** Provision for multi-field keys etc. goes beyond what is supported by `xsl:key`.

See Annotations (§3.13) for information on the role of the `{annotation}` property.

### 3.11.2 XML Representation of Identity-constraint Definition Schema Components

The XML representation for an identity-constraint definition schema component is either a `<key>`, a `<keyref>` or a `<unique>` element information item. The correspondences between the properties of those information items and properties of the component they correspond to are as follows:

**XML Representation Summary: unique Element Information Item**

```xml
<unique
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content:  (annotation?, (selector, field+))
</unique>
```

```xml
<key
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content:  (annotation?, (selector, field+))
</key>
```

```xml
<keyref
  id = ID
  name = NCName
  refer = QName
  {any attributes with non-schema namespace . . .}>
  Content:  (annotation?, (selector, field+))
</keyref>
```

3.11 Identity-constraint Definitions
Identity–constraint Definition Schema Component

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{name}</td>
<td>The ·actual value· of the name [attribute]</td>
</tr>
<tr>
<td>{target namespace}</td>
<td>The ·actual value· of the targetNamespace [attribute] of the parent schema element information item.</td>
</tr>
<tr>
<td>{identity–constraint category}</td>
<td>One of key, keyref or unique, depending on the item.</td>
</tr>
<tr>
<td>{selector}</td>
<td>A restricted XPath expression corresponding to the ·actual value· of the xpath [attribute] of the &lt;selector&gt; element information item among the [children]</td>
</tr>
<tr>
<td>{fields}</td>
<td>A sequence of XPath expressions, corresponding to the ·actual value·s of the xpath [attribute]s of the &lt;field&gt; element information item [children], in order.</td>
</tr>
<tr>
<td>{referenced key}</td>
<td>If the item is a &lt;keyref&gt;, the identity–constraint definition ·resolved· to by the ·actual value· of the refer [attribute], otherwise ·absent·.</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotation corresponding to the &lt;annotation&gt; element information item in the [children], if present, otherwise ·absent·.</td>
</tr>
</tbody>
</table>

Example

```xml
<xs:element name="vehicle">
  <xs:complexType>
    <xs:attribute name="plateNumber" type="xs:integer"/>
    <xs:attribute name="state" type="twoLetterCode"/>
  </xs:complexType>
</xs:element>

<xs:element name="state">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="vehicle" maxOccurs="unbounded"/>
      <xs:element ref="person" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

3.11 Identity–constraint Definitions
A state element is defined, which contains a code child and some vehicle and person children. A vehicle in turn has a plateNumber attribute, which is an integer, and a state attribute. State's codes are a key for them within the document. Vehicle's plateNumbers are a key for them within states, and state and plateNumber is asserted to be a key for vehicle within the document as a whole. Furthermore, a person element has an empty car child, with regState and regPlate attributes, which
are then asserted together to refer to vehicles via the carRef constraint. The requirement that a vehicle's state match its containing state's code is not expressed here.

### 3.11.3 Constraints on XML Representations of Identity–constraint Definitions

**Schema Representation Constraint: Identity–constraint Definition Representation OK**

In addition to the conditions imposed on `<key>`, `<keyref>` and `<unique>` element information items by the schema for schemas, the corresponding identity–constraint definition must satisfy the conditions set out in Constraints on Identity–constraint Definition Schema Components (§3.11.6).

### 3.11.4 Identity–constraint Definition Validation Rules

**Validation Rule: Identity–constraint Satisfied**

For an element information item to be locally _valid_ with respect to an identity–constraint _all_ of the following must be true:1 The {selector}, with the element information item as the context node, evaluates to a node–set (as defined in [XPath]). [Definition:] Call this the target node set .2 Each node in the target node set is an element node among the descendants of the context node.3 For each node in the target node set, all of the {fields}, with that node as the context node, evaluate to either an empty node–set or a node–set with exactly one member, which must have a simple type. [Definition:] Call the sequence of the type–determined values (as defined in [XML Schemas: Datatypes]) of the schema normalized value of the element and/or attribute information items in those node–sets in order the key–sequence of the node.4 [Definition:] Call the subset of the target node set, for which all the {fields} evaluate to a node–set with exactly one member which is an element or attribute node with a simple type the qualified node set. The appropriate case among the following must be true:4.1 If the {identity–constraint category} is unique, then no two members of the qualified node set have key–sequences whose members are pairwise equal, as defined by Equal in [XML Schemas: Datatypes].4.2 If the {identity–constraint category} is key, then all of the following must be true:4.2.1 The target node set and the qualified node set are equal, that is, every member of the target node set is also a member of the qualified node set and vice versa.4.2.2 No two members of the qualified node set have key–sequences whose members are pairwise equal, as defined by Equal in [XML Schemas: Datatypes].4.2.3 No element member of the key–sequence of any member of the qualified node set was assessed as valid by reference to an element declaration whose {nillable} is true.4.3 If the {identity–constraint category} is keyref, then for each member of the qualified node set (call this the keyref member), there must be a node table associated with the {referenced key} in the identity–constraint table of the element information item (see Identity–constraint Table (§3.11.5), which must be understood as logically prior to this clause of this constraint, below) and there must be an entry in that table whose key–sequence is equal to the keyref member’s key–sequence, member for member, as defined by Equal in [XML Schemas: Datatypes].

**NOTE:** The use of [schema normalized value] in the definition of key sequence above means that default or fixed value constraints may play a part in key sequence-s.

**NOTE:** Although this specification defines a post–schema–validation infoset contribution which would enable schema–aware processors to implement clause 4.2.3 above (Element Declaration (§3.3.5)), processors are not required to provide it. This clause can be read as if in the absence of this infoset contribution, the value of the relevant {nillable} property must be available.

### 3.11.5 Identity–constraint Definition Information Set Contributions

**Schema Information Set Contribution: Identity–constraint Table**

[Definition:] An eligible identity–constraint of an element information item is one such that clause 4.1 or
clause 4.2 of Identity–constraint Satisfied (§3.11.4) is satisfied with respect to that item and that constraint, or such that any of the element information item [children] of that item have an [identity–constraint table] property whose value has an entry for that constraint.

[Definition:] A node table is a set of pairs each consisting of a key–sequence and an element node.

Whenever an element information item has one or more eligible identity–constraints, in the post–schema–validation infoset that element information item has a property as follows: PSVI Contributions for element information items

[identity–constraint table]
one Identity–constraint Binding information item for each eligible identity–constraint, with properties as follows: PSVI Contributions for Identity–constraint Binding information items

[definition]
The eligible identity–constraint.

[node table]
A node table with one entry for every key–sequence, (call it k) and node (call it n) such that one of the following must be true:1 There is an entry in one of the node tables associated with the [definition] in an Identity–constraint Binding information item in at least one of the [identity–constraint table]s of the element information item [children] of the element information item whose key–sequence is k and whose node is n;2 n appears with key–sequence k in the qualified node set for the [definition], provided no two entries have the same key–sequence but distinct nodes. Potential conflicts are resolved by not including any conflicting entries which would have owed their inclusion to clause 1 above. Note that if all the conflicting entries arose under clause 1 above, this means no entry at all will appear for the offending key–sequence.

NOTE: The complexity of the above arises from the fact that keyref identity–constraints may be defined on domains distinct from the embedded domain of the identity–constraint they reference, or the domains may be the same but self–embedding at some depth. In either case the node table for the referenced identity–constraint needs to propagate upwards, with conflict resolution.

The Identity–constraint Binding information item, unlike others in this specification, is essentially an internal bookkeeping mechanism. It is introduced to support the definition of Identity–constraint Satisfied (§3.11.4) above. Accordingly, conformant processors may, but are not required to, expose them via [identity–constraint table] properties in the post–schema–validation infoset. In other words, the above constraints may be read as saying validation of identity–constraints proceeds as if such infoset items existed.

3.11.6 Constraints on Identity–constraint Definition Schema Components

All identity–constraint definitions (see Identity–constraint Definitions (§3.11)) must satisfy the following constraint.

Schema Component Constraint: Identity–constraint Definition Properties Correct
All of the following must be true:1 The values of the properties of an identity–constraint definition must be as described in the property tableau in The Identity–constraint Definition Schema Component (§3.11.1), modulo the impact of Missing Sub–components (§5.3).2 If the {identity–constraint category} is keyref, the cardinality of the {fields} must equal that of the {fields} of the {referenced key}. Schema Component Constraint: Selector Value OK
All of the following must be true: 1. The {selector} must be a valid XPath expression, as defined in [XPath]. 2. One of the following must be true: 2.1 It must conform to the following extended BNF:

### Selector XPath expressions

| Selector ::= | Path ( '|' Path )* |
|-------------|-------------------|
| Path ::=    | ('.//')? Step ( '/' Step )* |
| Step ::=    | '.' | NameTest |
| NameTest ::= | QName | '*' | NCName ':' '*' |

2.2 It must be an XPath expression involving the child axis whose abbreviated form is as given above.

**Schema Component Constraint: Fields Value OK**

All of the following must be true: 1. Each member of the {fields} must be a valid XPath expression, as defined in [XPath]. 2. One of the following must be true: 2.1 It must conform to the extended BNF given above for Selector, with the following modification:

### Path in Field XPath expressions

| Path ::= | ('.//')? ( Step '/' )* ( Step | '@' NameTest ) |

This production differs from the one above in allowing the final step to match an attribute node. 2.2 It must be an XPath expression involving the child and/or attribute axes whose abbreviated form is as given above.

### 3.12 Notation Declarations

3.12.1 The Notation Declaration Schema Component 3.12.2 XML Representation of Notation Declaration Schema Components 3.12.3 Constraints on XML Representations of Notation Declarations 3.12.4 Notation Declaration Validation Rules 3.12.5 Notation Declaration Information Set Contributions 3.12.6 Constraints on Notation Declaration Schema Components

Notation declarations reconstruct XML 1.0 NOTATION declarations.

Example

```xml
<x:s:notation name="jpeg" public="image/jpeg" system="viewer.exe"/>
```

The XML representation of a notation declaration.

### 3.12.1 The Notation Declaration Schema Component

The notation declaration schema component has the following properties:

- **Schema Component:** Notation Declaration
  - `{name}`: An NCName as defined by [XML–Namespaces].
  - `{target namespace}`: Either `absent` or a namespace name, as defined in [XML–Namespaces].
  - `{system identifier}`
Optional if `{public identifier}` is present. A URI reference.

{public identifier}
Optional if `{system identifier}` is present. A public identifier, as defined in [XML 1.0 (Second Edition)].

{annotation}
Optional. An annotation.

Notation declarations do not participate in ·validation· as such. They are referenced in the course of ·validating· strings as members of the `NOTATION` simple type.

See Annotations (§3.13) for information on the role of the `{annotation}` property.

### 3.12.2 XML Representation of Notation Declaration Schema Components

The XML representation for a notation declaration schema component is a `<notation>` element information item. The correspondences between the properties of that information item and properties of the component it corresponds to are as follows:

XML Representation Summary: `notation` Element Information Item

```xml
<notation
    id = ID
    name = NCName
    public = anyURI
    system = anyURI
    {any attributes with non-schema namespace . . .}>
    Content:  (annotation?)
</notation>
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{name}</code></td>
<td>The ·actual value· of the <code>name</code> [attribute]</td>
</tr>
<tr>
<td><code>{target namespace}</code></td>
<td>The ·actual value· of the <code>targetNamespace</code> [attribute] of the parent schema element information item.</td>
</tr>
<tr>
<td><code>{system identifier}</code></td>
<td>The ·actual value· of the <code>system</code> [attribute], if present, otherwise ·absent·.</td>
</tr>
<tr>
<td><code>{public identifier}</code></td>
<td>The ·actual value· of the <code>public</code> [attribute]</td>
</tr>
<tr>
<td><code>{annotation}</code></td>
<td>The annotation corresponding to the <code>&lt;annotation&gt;</code> element information item in the [children], if present, otherwise ·absent·.</td>
</tr>
</tbody>
</table>

Example

```xml
<xs:notation name="jpeg"
    public="image/jpeg" system="viewer.exe" />

<xs:element name="picture">
    <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="xs:hexBinary">
```

3.12 Notation Declarations
3.12.3 Constraints on XML Representations of Notation Declarations

**Schema Representation Constraint: Notation Definition Representation OK**
In addition to the conditions imposed on `<notation>` element information items by the schema for schemas, the corresponding notation definition must satisfy the conditions set out in Constraints on Notation Declaration Schema Components (§3.12.6).

3.12.4 Notation Declaration Validation Rules

None as such.

3.12.5 Notation Declaration Information Set Contributions

**Schema Information Set Contribution: Validated with Notation**
Whenever an attribute information item is valid with respect to a NOTATION, in the post-schema-validation infoset its parent element information item either has a property as follows: PSVI Contributions for element information items

- **[notation]**
  - An item isomorphic to the notation declaration whose `name` and `target namespace` match the `local name` and `namespace name` (as defined in QName Interpretation (§3.15.3)) of the attribute item's actual value.

- or has a pair of properties as follows: PSVI Contributions for element information items

- **[notation system]**
  - The value of the `system identifier` of that notation declaration.

- **[notation public]**
  - The value of the `public identifier` of that notation declaration.

**NOTE:** For compatibility, only one such attribute should appear on any given element. If more than one such attribute does appear, which one supplies the infoset property or properties above is not defined.
3.12.6 Constraints on Notation Declaration Schema Components

All notation declarations (see Notation Declarations (§3.12)) must satisfy the following constraint.

**Schema Component Constraint: Notation Declaration Correct**
The values of the properties of a notation declaration must be as described in the property tableau in The Notation Declaration Schema Component (§3.12.1), modulo the impact of Missing Sub–components (§5.3).

3.13 Annotations

3.13.1 The Annotation Schema Component

Annotations provide for human– and machine–targeted annotations of schema components.

Example

```xml
<xs:simpleType fn:note="special">
  <xs:annotation>
    <xs:documentation>A type for experts only</xs:documentation>
    <xs:appinfo>
      <fn:specialHandling>checkForPrimes</fn:specialHandling>
    </xs:appinfo>
  </xs:annotation>
</xs:simpleType>
```

XML representations of three kinds of annotation.

3.13.1 The Annotation Schema Component

The annotation schema component has the following properties:

**Schema Component: Annotation**

- **{application information}**
  A sequence of element information items.
- **{user information}**
  A sequence of element information items.
- **{attributes}**
  A sequence of attribute information items.

{user information} is intended for human consumption, {application information} for automatic processing. In both cases, provision is made for an optional URI reference to supplement the local information, as the value of the `source` attribute of the respective element information items. *Validation* does *not* involve dereferencing these URIs, when present. In the case of {user information}, indication should be given as to the identity of the (human) language used in the contents, using the `xml:lang` attribute.

{attributes} ensures that when schema authors take advantage of the provision for adding attributes from namespaces other than the XML Schema namespace to schema documents, they are available within the components corresponding to the element items where such attributes appear.
Annotations do not participate in validation as such. Provided an annotation itself satisfies all relevant Schema Component Constraints, it cannot affect the validation of element information items.

### 3.13.2 XML Representation of Annotation Schema Components

Annotation of schemas and schema components, with material for human or computer consumption, is provided for by allowing application information and human information at the beginning of most major schema elements, and anywhere at the top level of schemas. The XML representation for an annotation schema component is an `<annotation>` element information item. The correspondences between the properties of that information item and properties of the component it corresponds to are as follows:

**XML Representation Summary:** *annotation* Element Information Item

```
<annotation
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (appinfo | documentation)*
</annotation>
```

```
<appinfo
  source = anyURI>
  Content: (any)*
</appinfo>
```

```
<documentation
  source = anyURI
  xml:lang = language>
  Content: (any)*
</documentation>
```

**Annotation Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{application information}</td>
<td>A sequence of the <code>&lt;appinfo&gt;</code> element information items from among the [children], in order, if any, otherwise the empty sequence.</td>
</tr>
<tr>
<td>{user information}</td>
<td>A sequence of the <code>&lt;documentation&gt;</code> element information items from among the [children], in order, if any, otherwise the empty sequence.</td>
</tr>
<tr>
<td>{attributes}</td>
<td>A sequence of attribute information items, namely those allowed by the attribute wildcard in the type definition for the <code>&lt;annotation&gt;</code> item itself or for the enclosing items which correspond to the component within which the annotation component is located.</td>
</tr>
</tbody>
</table>

The annotation component corresponding to the `<annotation>` element in the example above will have one element item in each of its `{user information}` and `{application information}` and one attribute item in its `{attributes}`.
3.13.3 Constraints on XML Representations of Annotations

Schema Representation Constraint: Annotation Definition Representation OK
In addition to the conditions imposed on <annotation> element information items by the schema for schemas, the corresponding annotation must satisfy the conditions set out in Constraints on Annotation Schema Components (§3.13.6).

3.13.4 Annotation Validation Rules

None as such.

3.13.5 Annotation Information Set Contributions

None as such: the addition of annotations to the post-schema-validation infoset is covered by the post-schema-validation infoset contributions of the enclosing components.

3.13.6 Constraints on Annotation Schema Components

All annotations (see Annotations (§3.13)) must satisfy the following constraint.

Schema Component Constraint: Annotation Correct
The values of the properties of an annotation must be as described in the property tableau in The Annotation Schema Component (§3.13.1), modulo the impact of Missing Sub-components (§5.3).

3.14 Simple Type Definitions

3.14.1 (non-normative) The Simple Type Definition Schema Component
3.14.2 (non-normative) XML Representation of Simple Type Definition Schema Components
3.14.3 (non-normative) Constraints on XML Representations of Simple Type Definitions
3.14.4 Simple Type Definition Validation Rules
3.14.5 Simple Type Definition Information Set Contributions
3.14.6 Constraints on Simple Type Definition Schema Components
3.14.7 Built-in Simple Type Definition

NOTE: This section consists of a combination of non-normative versions of normative material from [XML Schemas: Datatypes], for local cross-reference purposes, and normative material relating to the interface between schema components defined in this specification and the simple type definition component.

Simple type definitions provide for constraining character information item [children] of element and attribute information items.

Example

```xml
<xs:simpleType name="farenheitWaterTemp">
  <xs:restriction base="xs:number">
    <xs:_fractionDigits value="2"/>
    <xs:minExclusive value="0.00"/>
    <xs:maxExclusive value="100.00"/>
  </xs:restriction>
</xs:simpleType>
```

The XML representation of a simple type definition.
3.14.1 (non−normative) The Simple Type Definition Schema Component

The simple type definition schema component has the following properties:

Schema Component: Simple Type Definition

{name}
Optional. An NCName as defined by [XML−Namespaces].

{target namespace}
Either ·absent·, or a namespace name, as defined in [XML−Namespaces].

{base type definition}
A simple type definition, which may be the ·simple ur−type definition·.

{facets}
A set of constraining facets.

{fundamental facets}
A set of fundamental facets.

{final}
A subset of {extension, list, restriction, union}.

{variety}
One of {atomic, list, union}. Depending on the value of {variety}, further properties are defined as follows:

atomic

{primitive type definition}
A built−in primitive simple type definition (or the ·simple ur−type definition·).

list

{item type definition}
A simple type definition.

union

{member type definitions}
A non−empty sequence of simple type definitions.

{annotation}
Optional. An annotation.

Simple types are identified by their {name} and {target namespace}. Except for anonymous simple types (those with no {name}), since type definitions (i.e. both simple and complex type definitions taken together) must be uniquely identified within an -XML Schema-, no simple type definition can have the same name as another simple or complex type definition. Simple type {name}s and {target namespace}s are provided for reference from instances (see xsi:type (§2.6.1)), and for use in the XML representation of schema components (specifically in <element> and <attribute>). See References to schema components across namespaces (§4.2.3) for the use of component identifiers when importing one schema into another.

NOTE: The {name} of a simple type is not ipso facto the [(local) name] of the element or attribute information items -validated- by that definition. The connection between a name and a type definition is described in Element Declarations (§3.3) and Attribute Declarations (§3.2).

A simple type definition with an empty specification for {final} can be used as the {base type definition} for other types derived by either of extension or restriction, or as the {item type definition} in the definition of a list, or in the {member type definitions} of a union; the explicit values extension, restriction, list and union prevent further derivations by extension (to yield a complex type) and restriction (to yield a simple type) and use in constructing lists and unions respectively.
{variety} determines whether the simple type corresponds to an atomic, list or union type as defined by [XML Schemas: Datatypes].

As described in Type Definition Hierarchy (§2.2.1.1), every simple type definition is a restriction of some other simple type (the base type definition), which is the simple ur-type definition; if and only if the type definition in question is one of the built-in primitive datatypes, or a list or union type definition. Each atomic type is ultimately a restriction of exactly one such built-in simple primitive type definition.

{facets} for each simple type definition are selected from those defined in [XML Schemas: Datatypes]. For atomic definitions, these are restricted to those appropriate for the corresponding primitive type definition. Therefore, the value space and lexical space (i.e. what is validated by any atomic simple type) is determined by the pair (primitive type definition, {facets}).

As specified in [XML Schemas: Datatypes], list simple type definitions validate space separated tokens, each of which conforms to a specified simple type definition, the item type definition. The item type specified must not itself be a list type, and must be one of the types identified in [XML Schemas: Datatypes] as a suitable item type for a list simple type. In this case the {facets} apply to the list itself, and are restricted to those appropriate for lists.

A union simple type definition validates strings which satisfy at least one of its member type definitions. As in the case of list, the {facets} apply to the union itself, and are restricted to those appropriate for unions.

As discussed in Type Definition Hierarchy (§2.2.1.1), the ur-type definition functions as a simple type when used as the base type definition for the built-in primitive datatypes and for list and union type definitions. It is considered to have an unconstrained lexical space, and a value space consisting of the union of the value spaces of all the built-in primitive datatypes and the set of all lists of all members of the value spaces of all the built-in primitive datatypes.

The simple ur-type definition must not be named as the base type definition of any user-defined simple types: as it has no constraining facets, this would be incoherent.

See Annotations (§3.13) for information on the role of the {annotation} property.

3.14.2 (non-normative) XML Representation of Simple Type Definition Schema Components

NOTE: This section reproduces a version of material from [XML Schemas: Datatypes], for local cross-reference purposes.

XML Representation Summary: simpleType Element Information Item

```xml
<simpleType
    final = (#all | list | union | restriction))
    id = ID
    name = NCName
    {any attributes with non-schema namespace . . .}>
    {Content: (annotation?, (restriction | list | union))}
</simpleType>
```

```xml
<restriction
    base = QName
    id = ID
```

3.14 Simple Type Definitions
Simple Type Definition Schema Component

Property | Representation
---|---
{name} | The _actual value_ of the name [attribute] if present, otherwise _absent_.
{target namespace} | The _actual value_ of the targetNamespace [attribute] of the <schema> ancestor element information item if present, otherwise _absent_.
{base type definition} | The appropriate case among the following: 1 If the <restriction> alternative is chosen, then the type definition _resolved_ by the _actual value_ of the base [attribute] of <restriction>, if present, otherwise the type definition corresponding to the <simpleType> among the [children] of <restriction>. 2 If the <list> or <union> alternative is chosen, then the _simple ur-type definition_.
{final} | As for the {prohibited substitutions} property of complex type definitions, but using the final and finalDefault [attributes] in place of the block and blockDefault [attributes] and with the relevant set being {extension, restriction, list, union}.
{variety} | If the <list> alternative is chosen, then list, otherwise if the <union> alternative is chosen, then union, otherwise (the <restriction> alternative is chosen), then the {variety} of the {base type definition}.

If the {variety} is atomic, the following additional property mappings also apply:

Atomic Simple Type Definition Schema Component

Property | Representation
---|---
{primitive type definition} | The built-in primitive type definition from which the {base type definition} is derived.
{facets} | A set of facet components _constituting a restriction_ of the {facets} of the {base type definition} with respect to a set of facet components corresponding to the appropriate element information items among the [children] of <restriction> (i.e. those which specify
facets, if any), as defined in Simple Type Restriction (Facets) (§3.14.3).

If the {variety} is list, the following additional property mappings also apply:

**List Simple Type Definition Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{item type definition}</td>
<td>The appropriate case among the following: 1 If the &lt;list&gt; alternative is chosen, then the type definition resolved, to the actual value of the itemType attribute of &lt;list&gt;, if present, otherwise the type definition corresponding to the &lt;simpleType&gt; among the [children] of &lt;list&gt;. 2 If the &lt;restriction&gt; option is chosen, then the {item type definition} of the {base type definition}.</td>
</tr>
<tr>
<td>{facets}</td>
<td>If the &lt;restriction&gt; alternative is chosen, a set of facet components constituting a restriction of the {facets} of the {base type definition} with respect to a set of facet components corresponding to the appropriate element information items among the [children] of &lt;restriction&gt; (i.e. those which specify facets, if any), as defined in Simple Type Restriction (Facets) (§3.14.3), otherwise the empty set.</td>
</tr>
</tbody>
</table>

If the {variety} is union, the following additional property mappings also apply:

**Union Simple Type Definition Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{member type definitions}</td>
<td>The appropriate case among the following: 1 If the &lt;union&gt; alternative is chosen, then [Definition:] define the explicit members as the type definitions resolved, to the items in the actual value of the memberTypes attribute, if any, followed by the type definitions corresponding to the &lt;simpleType&gt;s among the [children] of &lt;union&gt;, if any. The actual value is then formed by replacing any union type definition in the explicit members with the members of their {member type definitions}, in order. 2 If the &lt;restriction&gt; option is chosen, then the {member type definitions} of the {base type definition}.</td>
</tr>
<tr>
<td>{facets}</td>
<td>If the &lt;restriction&gt; alternative is chosen, a set of facet components constituting a restriction of the {facets} of the {base type definition} with respect to a set of facet components corresponding to the appropriate element information items among the [children] of &lt;restriction&gt; (i.e. those which specify facets, if any), as defined in Simple Type Restriction (Facets) (§3.14.3), otherwise the empty set.</td>
</tr>
</tbody>
</table>

3.14.3 (non−normative) Constraints on XML Representations of Simple Type Definitions

**Schema Representation Constraint: Simple Type Definition Representation OK**

In addition to the conditions imposed on <simpleType> element information items by the schema for schemas, all of the following must be true: 1 The corresponding simple type definition, if any, must satisfy the conditions set out in Constraints on Simple Type Definition Schema Components (§3.14.6). 2 If the <restriction> alternative is chosen, either it must have a base [attribute] or a <simpleType> among its [children], but not both. 3 If the <list> alternative is chosen, either it must have an itemType [attribute] or a <simpleType> among its [children], but not both. 4 Circular union type definition is disallowed. That is, if the <union> alternative is chosen, there must not be any entries in the memberTypes [attribute] at any depth which resolve to the component corresponding to the <simpleType>. **Schema Representation Constraint: Simple Type Restriction (Facets)**

For a simple type definition (call it R) to restrict another simple type definition (call it B) with a set of facets (call this S) all of the following must be true: 1 The {variety} and {primitive type definition} of R are the same as those of B. 2 The {facets} of R are the union of S and the {facets} of B, eliminating duplicates. To
eliminate duplicates, when a facet of the same kind occurs in both $S$ and the {facets} of $B$, the one in the {facets} of $B$ is not included, with the exception of enumeration and pattern facets, for which multiple occurrences with distinct values are allowed.

[Definition:] If clause 2 above holds, the {facets} of $R$ constitute a restriction of the {facets} of $B$ with respect to $S$.

### 3.14.4 Simple Type Definition Validation Rules

**Validation Rule: String Valid**

A string is locally valid with respect to a simple type definition if it is schema-valid with respect to that definition as defined by Datatype Valid in [XML Schemas: Datatypes].

### 3.14.5 Simple Type Definition Information Set Contributions

None as such.

### 3.14.6 Constraints on Simple Type Definition Schema Components

All simple type definitions (see Simple Type Definitions (§3.14)) must satisfy the following constraints.

**Schema Component Constraint: Simple Type Definition Properties Correct**

All of the following must be true:

1. The values of the properties of a simple type definition must be as described in the property tableau in Datatype definition, modulo the impact of Missing Sub–components (§5.3).
2. Circular definitions are disallowed. That is, it must be possible to reach a built–in primitive datatype or the -simple ur–type definition- by repeatedly following the [base type definition].
3. The [final] of the [base type definition] must not contain restriction.
4. If the [base type definition] is not the -simple ur–type definition-, all of the following must be true:
   1. The definition must be a valid restriction as defined in Derivation Valid (Restriction, Simple) (§3.14.6).
   2. If {variety} is not atomic, then the appropriate case among the following must be true:
      1. If the {variety} is list, then the {final} of the [base type definition] must not contain list.
      2. If the {variety} is union, then the {final} of the [base type definition] must not contain union.

**Schema Component Constraint: Derivation Valid (Restriction, Simple)**

The appropriate case among the following must be true:

1. If the {variety} is atomic, then all of the following must be true:
   1.1 The [base type definition] must be an atomic simple type definition or a built–in primitive datatype.
   1.2 The [final] of the [base type definition] must not contain restriction.
   1.3 For each facet in the {facets} there must be a facet of the same kind in the {facets} of the [base type definition] of whose {value} the facet in question's {value} must be a valid restriction as defined in [XML Schemas: Datatypes].
2. If the {variety} is list, then all of the following must be true:
   2.1 The [item type definition] must have a {variety} of atomic or union (in which case all the [member type definitions] must be atomic).
   2.2 Only length, minLength, maxLength, pattern and enumeration facet components are allowed among the {facets}.
3. If the [base type definition] is not the -simple ur–type definition-, then all of the following must be true:
   3.1 The [base type definition] must have a {variety} of list.
   3.2 The [final] of the [base type definition] must not contain restriction.
   3.3 For each facet in the {facets} there must be a facet of the same kind in the {facets} of the [base type definition] of whose {value} the facet in question's {value} must be a valid restriction as defined in [XML Schemas: Datatypes].
4. If the {variety} is union, then all of the following must be true:
   4.1 The [member type definitions] must all have {variety} of atomic or list.
   4.2 Only pattern and enumeration facet components are allowed among the {facets}.
   4.3 If the [base type definition] is not the -simple ur–type definition-, then all of the following must be true:
      4.3.1 The [base type definition] must have a {variety} of union.
      4.3.2 The [final] of the [base type definition] must not contain restriction.
      4.3.3 For each facet in the {facets} there must be a facet of the same kind in the {facets} of the [base type definition] of whose {value} the facet in question's {value} must be a valid restriction as defined in [XML Schemas: Datatypes].
If this constraint Derivation Valid (Restriction, Simple) (§3.14.6) holds of a simple type definition, it is a valid restriction of its base type definition.

The following constraint defines relations appealed to elsewhere in this specification.

**Schema Component Constraint: Type Derivation OK (Simple)**

For a simple type definition (call it D, for derived) to be validly derived from a simple type definition (call this B, for base) given a subset of \{extension, restriction, list, union\} (of which only restriction is actually relevant) one of the following must be true:

1. They are the same type definition.
2. All of the following must be true:
   1. restriction is not in the subset, or in the \{final\} of its own \{base type definition\}.
   2. One of the following must be true:
      1. D's \{base type definition\} is not the \{simple ur−type definition\} and is validly derived from B given the subset, as defined by this constraint.
      2. B's \{variety\} is list or union and D is validly derived from a type definition in B's \{member type definitions\}, given the subset, as defined by this constraint.

3.14.7 Built−in Simple Type Definition

There is a simple type definition nearly equivalent to the simple version of the \{ur−type definition\}, present in every schema by definition. It has the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{name}</td>
<td>anySimpleType</td>
</tr>
<tr>
<td>{target namespace}</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
</tr>
<tr>
<td>{base type definition}</td>
<td>·the ur−type definition·</td>
</tr>
<tr>
<td>{final}</td>
<td>The empty set</td>
</tr>
<tr>
<td>{variety}</td>
<td>·absent·</td>
</tr>
</tbody>
</table>

Simple type definitions for all the built−in primitive datatypes, namely string, boolean, float, double, number, dateTime, duration, time, date, gMonth, gMonthDay, gDay, gYear, gYearMonth, hexBinary, base64Binary, anyURI (see the Primitive Datatypes section of [XML Schemas: Datatypes]), as well as for the simple and complex \{ur−type definitions\}, (as previously described), are present by definition in every schema. All are in the XML Schema \{target namespace\} (namespace name http://www.w3.org/2001/XMLSchema), have an atomic \{variety\} with an empty \{facets\} and the simple \{ur−type definition\}, as their \{base type definition\}, and themselves as \{primitive type definition\}.

Similarly, simple type definitions for all the built−in derived datatypes (see the Derived Datatypes section of [XML Schemas: Datatypes]) are present by definition in every schema, with properties as specified in [XML Schemas: Datatypes] and as represented in XML in Schema for Schemas (normative) (§A).

3.15 Schemas as a Whole

3.15.1 The Schema Itself
3.15.2 XML Representations of Schemas
3.15.3 Constraints on XML Representations of Schemas
3.15.4 Validation Rules for Schemas as a Whole
3.15.5 Schema Information Set Contributions
3.15.6 Constraints on Schemas as a Whole
A schema consists of a set of schema components.

Example

<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.example.com/example">
   ...
</xs:schema>

The XML representation of the skeleton of a schema.

3.15.1 The Schema Itself

At the abstract level, the schema itself is just a container for its components.

Schema Component: Schema

| {type definitions} | A set of named simple and complex type definitions. |
| {attribute declarations} | A set of named (top−level) attribute declarations. |
| {element declarations} | A set of named (top−level) element declarations. |
| {attribute group definitions} | A set of named attribute group definitions. |
| {model group definitions} | A set of named model group definitions. |
| {notation declarations} | A set of notation declarations. |
| {annotations} | A set of annotations. |

3.15.2 XML Representations of Schemas

A schema is represented in XML by one or more schema documents, that is, one or more <schema> element information items. A schema document contains representations for a collection of schema components, e.g. type definitions and element declarations, which have a common {target namespace}. A schema document which has one or more <import> element information items corresponds to a schema with components with more than one {target namespace}, see Import Constraints and Semantics (§4.2.3).

XML Representation Summary: schema Element Information Item

<schema
   attributeFormDefault = (qualified | unqualified) : unqualified
   blockDefault = (#all | List of (extension | restriction | substitution)) : "
   elementFormDefault = (qualified | unqualified) : unqualified
   finalDefault = (#all | List of (extension | restriction)) : "
   id = ID
   targetNamespace = anyURI
   version = token
   xml:lang = language
The simple and complex type definitions corresponding to all the `<simpleType>` and `<complexType>` element information items in the `[children]`, if any, plus any included or imported definitions, see Assembling a schema for a single target namespace from multiple schema definition documents (§4.2.1) and References to schema components across namespaces (§4.2.3).

The (top–level) attribute declarations corresponding to all the `<attribute>` element information items in the `[children]`, if any, plus any included or imported declarations, see Assembling a schema for a single target namespace from multiple schema definition documents (§4.2.1) and References to schema components across namespaces (§4.2.3).

The (top–level) element declarations corresponding to all the `<element>` element information items in the `[children]`, if any, plus any included or imported declarations, see Assembling a schema for a single target namespace from multiple schema definition documents (§4.2.1) and References to schema components across namespaces (§4.2.3).

The attribute group definitions corresponding to all the `<attributeGroup>` element information items in the `[children]`, if any, plus any included or imported definitions, see Assembling a schema for a single target namespace from multiple schema definition documents (§4.2.1) and References to schema components across namespaces (§4.2.3).

The model group definitions corresponding to all the `<group>` element information items in the `[children]`, if any, plus any included or imported definitions, see Assembling a schema for a single target namespace from multiple schema definition documents (§4.2.1) and References to schema components across namespaces (§4.2.3).

The notation declarations corresponding to all the `<notation>` element information items in the `[children]`, if any, plus any included or imported declarations, see Assembling a schema for a single target namespace from multiple schema definition documents (§4.2.1) and References to schema components across namespaces (§4.2.3).

The annotations corresponding to all the `<annotation>` element information items in the `[children]`, if any.

Note that none of the attribute information items displayed above correspond directly to properties of schemas. The `blockDefault`, `finalDefault`, `attributeFormDefault`, `elementFormDefault` and `targetNamespace` attributes are appealed to in the sub–sections above, as they provide global information applicable to many representation/component correspondences. The other attributes (`id` and `version`) are for user convenience, and this specification defines no semantics for them.
The definition of the schema abstract data model in XML Schema Abstract Data Model (§2.2) makes clear that most components have a {target namespace}. Most components corresponding to representations within a given <schema> element information item will have a {target namespace} which corresponds to the targetNamespace attribute.

Since the empty string is not a legal namespace name, supplying an empty string for targetNamespace is incoherent, and is not the same as not specifying it at all. The appropriate form of schema document corresponding to a schema whose components have no {target namespace} is one which has no targetNamespace attribute specified at all.

NOTE: The XML namespaces Recommendation discusses only instance document syntax for elements and attributes; it therefore provides no direct framework for managing the names of type definitions, attribute group definitions, and so on. Nevertheless, the specification applies the target namespace facility uniformly to all schema components, i.e. not only declarations but also definitions have a {target namespace}.

Although the example schema at the beginning of this section might be a complete XML document, <schema> need not be the document element, but can appear within other documents. Indeed there is no requirement that a schema correspond to a (text) document at all: it could correspond to an element information item constructed 'by hand', for instance via a DOM−conformant API.

Aside from <include> and <import>, which do not correspond directly to any schema component at all, each of the element information items which may appear in the content of <schema> corresponds to a schema component, and all except <annotation> are named. The sections below present each such item in turn, setting out the components to which it may correspond.

3.15.2.1 References to Schema Components

Reference to schema components from a schema document is managed in a uniform way, whether the component corresponds to an element information item from the same schema document or is imported (References to schema components across namespaces (§4.2.3)) from an external schema (which may, but need not, correspond to an actual schema document). The form of all such references is a QName.

[Definition:] A QName is a name with an optional namespace qualification, as defined in [XML−Namespaces]. When used in connection with the XML representation of schema components or references to them, this refers to the simple type QName as defined in [XML Schemas: Datatypes].

[Definition:] An NCName is a name with no colon, as defined in [XML−Namespaces]. When used in connection with the XML representation of schema components in this specification, this refers to the simple type NCName as defined in [XML Schemas: Datatypes].

In each of the XML representation expositions in the following sections, an attribute is shown as having type QName if and only if it is interpreted as referencing a schema component.

Example

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xhtml="http://www.w3.org/1999/xhtml"
    xmlns="http://www.example.com"
    targetNamespace="http://www.example.com">
    ...
</xs:schema>
```
The first of these is most probably a local reference, i.e. a reference to a type definition corresponding to a `<complexType>` element information item located elsewhere in the schema document, the other two refer to type definitions from schemas for other namespaces and assume that their namespaces have been declared for import. See References to schema components across namespaces (§4.2.3) for a discussion of importing.

3.15.2.2 References to Schema Components from Elsewhere

The names of schema components such as type definitions and element declarations are not of type ID: they are not unique within a schema, just within a symbol space. This means that simple fragment identifiers will not always work to reference schema components from outside the context of schema documents.

There is currently no provision in the definition of the interpretation of fragment identifiers for the text/xml MIME type, which is the MIME type for schemas, for referencing schema components as such. However, [XPointer] provides a mechanism which maps well onto the notion of symbol spaces as it is reflected in the XML representation of schema components. A fragment identifier of the form 
#xpointer(xs:schema/xs:element[@name="person"])
will uniquely identify the representation of a top–level element declaration with name person, and similar fragment identifiers can obviously be constructed for the other global symbol spaces.

Short–form fragment identifiers may also be used in some cases, that is when a DTD or XML Schema is available for the schema in question, and the provision of an id attribute for the representations of all primary and secondary schema components, which is of type ID, has been exploited.

It is a matter for applications to specify whether they interpret document–level references of either of the above varieties as being to the relevant element information item (i.e. without special recognition of the relation of schema documents to schema components) or as being to the corresponding schema component.

3.15.3 Constraints on XML Representations of Schemas

Schema Representation Constraint: QName Interpretation

Where the type of an attribute information item in a document involved in validation is identified as QName, its actual value is composed of a [Definition:] local name and a [Definition:] namespace name. Its actual value is determined based on its normalized value and the containing element information item's [in–scope namespaces] following [XML–Namespaces]:

The appropriate case among the following must be true:

1. If its normalized value is prefixed, then all of the following must be true: 1.1 There must be a namespace in the [in–scope namespaces] whose [prefix] matches the prefix. 1.2 Its namespace name is the [namespace name] of that namespace.
2. Otherwise (its normalized value is unprefixed) all of the following must be true: 2.1 Its local name is its normalized value.
3. The appropriate case among the following must be true: 2.2 1 If there is a namespace in the [in–scope namespaces] whose [prefix] has no value, then its namespace name is the [namespace name] of that namespace. 2.2.2 Otherwise its namespace name
name is absent.

In the absence of the \texttt{[in−scope namespaces]} property in the infoset for the schema document in question, processors must reconstruct equivalent information as necessary, using the \texttt{[namespace attributes]} of the containing element information item and its ancestors.

[Definition:] Whenever the word \underline{resolve} in any form is used in this chapter in connection with a \texttt{QName} in a schema document, the following definition \underline{QName resolution (Schema Document)} (§3.15.3) should be understood:

**Schema Representation Constraint: QName resolution (Schema Document)**

For a \texttt{QName} to resolve to a schema component of a specified kind all of the following must be true: 1

1. If the kind specified is simple or complex type definition, then the property is the \texttt{[type definitions]}. If the kind specified is attribute declaration, then the property is the \texttt{[attribute declarations]}. 1.3 If the kind specified is element declaration, then the property is the \texttt{[element declarations]}. 1.4 If the kind specified is attribute group, then the property is the \texttt{[attribute group definitions]}. 1.5 If the kind specified is model group, then the property is the \texttt{[model group definitions]}. 1.6 If the kind specified is notation declaration, then the property is the \texttt{[notation declarations]}.

2. Its \underline{local name} matches the \underline{local name} of the \texttt{QName}.

3. Its \underline{target namespace} is identical to the \underline{namespace name} of the \texttt{QName}.

4. Its \underline{namespace name} is either the target namespace of the schema document containing the \texttt{QName} or that schema document contains an \texttt{<import>} element information item the \underline{actual value} of whose \underline{namespace} \underline{attribute} is identical to that \underline{namespace name}.

3.15.4 Validation Rules for Schemas as a Whole

As the discussion above at Schema Component Details (§3) makes clear, at the level of schema components and \underline{validation}, reference to components by name is normally not involved. In a few cases, however, qualified names appearing in information items being \underline{validated} must be resolved to schema components by such lookup. The following constraint is appealed to in these cases.

**Validation Rule: QName resolution (Instance)**

A pair of a local name and a namespace name (or \underline{absent}) resolve to a schema component of a specified kind in the context of \underline{validation} by appeal to the appropriate property of the schema being used for the \underline{assessment}. Each such property indexes components by name. The property to use is determined by the kind of component specified, that is, the appropriate case among the following must be true: 1

1. If the kind specified is simple or complex type definition, then the property is the \texttt{[type definitions]}. 2. If the kind specified is attribute declaration, then the property is the \texttt{[attribute declarations]}. 3. If the kind specified is element declaration, then the property is the \texttt{[element declarations]}. 4. If the kind specified is attribute group, then the property is the \texttt{[attribute group definitions]}. 5. If the kind specified is model group, then the property is the \texttt{[model group definitions]}. 6. If the kind specified is notation declaration, then the property is the \texttt{[notation declarations]}. The component resolved to is the entry in the table whose \underline{local name} matches the local name of the pair and whose \underline{target namespace} is identical to the namespace name of the pair.

3.15.5 Schema Information Set Contributions

**Schema Information Set Contribution: Schema Information**

Schema components provide a wealth of information about the basis of \underline{assessment}, which may well be of relevance to subsequent processing. Reflecting component structure into a form suitable for inclusion in the post−schema−validation infoset is the way this specification provides for making this information available.
Accordingly, [Definition:] by an **item isomorphic** to a component is meant an information item whose type is equivalent to the component's, with one property per property of the component, with the same name, and value either the same atomic value, or an information item corresponding in the same way to its component value, recursively, as necessary.

Processors must add a property in the post-schema-validation infoset to the element information item at which `-assessment-` began, as follows: **PSVI Contributions for element information items**

**[schema information]**

A set of **namespace schema information** information items, one for each namespace name which appears as the `{target namespace}` of any schema component in the schema used for that assessment, and one for `-absent-` if any schema component in the schema had no `{target namespace}`. Each **namespace schema information** information item has the following properties and values: **PSVI Contributions for namespace schema information information items**

**[schema namespace]**

A namespace name or `-absent-`.

**[schema components]**

A (possibly empty) set of schema component information items, each one an **item isomorphic** to a component whose `{target namespace}` is the sibling `[schema namespace]` property above, drawn from the schema used for `-assessment-`.

**[schema documents]**

A (possibly empty) set of **schema document** information items, with properties and values as follows, for each schema document which contributed components to the schema, and whose `targetNamespace` matches the sibling `[schema namespace]` property above (or whose `targetNamespace` was `-absent-` but that contributed components to that namespace by being `<include>`d by a schema document with that `targetNamespace` as per Assembling a schema for a single target namespace from multiple schema definition documents (§4.2.1)): **PSVI Contributions for schema document information items**

**[document location]**

Either a URI reference, if available, otherwise `-absent-`.

**[document]**

A document information item, if available, otherwise `-absent-`.

The `[schema components]` property is provided for processors which wish to provide a single access point to the components of the schema which was used during `-assessment-`. Lightweight processors are free to leave it empty, but if it is provided, it must contain at a minimum all the top-level (i.e. named) components which actually figured in the `-assessment-`, either directly or (because an anonymous component which figured is contained within) indirectly. **Schema Information Set Contribution: ID/IDREF Table**

In the post-schema-validation infoset a set of **ID/IDREF binding** information items is associated with the `-validation root-` element information item: **PSVI Contributions for element information items**

**[ID/IDREF table]**

A (possibly empty) set of **ID/IDREF binding** information items, as specified below.

[Definition:] Let the **eligible item set** be the set of consisting of every attribute or element information item for which all of the following are true1 its `[validation context]` is the `-validation root-`:2 it was successfully `-validated-`, with respect to an attribute declaration as per Attribute Locally Valid (§3.2.4) or element declaration as per Element Locally Valid (Element) (§3.3.4) (as appropriate) whose attribute `{type definition}` or element `{type definition}` (respectively) is the built-in **ID**, **IDREF** or **IDREFS** simple type definition or a type derived from one of them.

3.15 Schemas as a Whole 158
Then there is one **ID/IDREF binding** in the [ID/IDREF table] for every distinct string which is one of the following: 1 the ·actual value· of a member of the ·eligible item set· whose type definition is or is derived from ID or IDREF; 2 one of the items in the ·actual value· of a member of the ·eligible item set· whose type definition is or is derived from IDREFS. Each **ID/IDREF binding** has properties as follows: PSVI Contributions for ID/IDREF binding information items

**[id]**

The string identified above.

**[binding]**

A set consisting of every element information item for which all of the following are true 1 its ·validation context· is the ·validation root·; 2 it has an attribute information item in its [attributes] or an element information item in its [children] which was ·validated· by the built−in ID simple type definition or a type derived from it whose [schema normalized value] is the [id] of this **ID/IDREF binding**.

The net effect of the above is to have one entry for every string used as an id, whether by declaration or by reference, associated with those elements, if any, which actually purport to have that id. See Validation Root Valid (ID/IDREF) (§3.3.4) above for the validation rule which actually checks for errors here.

**NOTE:** The **ID/IDREF binding** information item, unlike most other aspects of this specification, is essentially an internal bookkeeping mechanism. It is introduced to support the definition of Validation Root Valid (ID/IDREF) (§3.3.4) above. Accordingly, conformant processors may, but are not required to, expose it in the post−schema−validation infoset. In other words, the above constraint may be read as saying ·assessment· proceeds as if such an infoset item existed.

### 3.15.6 Constraints on Schemas as a Whole

All schemas (see Schemas as a Whole (§3.15)) must satisfy the following constraint.

**Schema Component Constraint: Schema Properties Correct**

All of the following must be true: 1 The values of the properties of a schema must be as described in the property tableau in The Schema Itself (§3.15.1), modulo the impact of Missing Sub−components (§5.3); 2 Each of the {type definitions}, {element declarations}, {attribute group definitions}, {model group definitions} and {notation declarations} must not contain two or more schema components with the same {name} and {target namespace}.

### 4 Schemas and Namespaces: Access and Composition

This chapter defines the mechanisms by which this specification establishes the necessary precondition for ·assessment·, namely access to one or more schemas. This chapter also sets out in detail the relationship between schemas and namespaces, as well as mechanisms for modularization of schemas, including provision for incorporating definitions and declarations from one schema in another, possibly with modifications.

Conformance (§2.4) describes three levels of conformance for schema processors, and Schemas and Schema−validity Assessment (§5) provides a formal definition of ·assessment·. This section sets out in detail the 3−layer architecture implied by the three conformance levels. The layers are:

1. The ·assessment· core, relating schema components and instance information items;
2. Schema representation: the connections between XML representations and schema components,
including the relationships between namespaces and schema components;

3. XML Schema web-interoperability guidelines: instance->schema and schema->schema connections for the WWW.

Layer 1 specifies the manner in which a schema composed of schema components can be applied to in the assessment of an instance element information item. Layer 2 specifies the use of <schema> elements in XML documents as the standard XML representation for schema information in a broad range of computer systems and execution environments. To support interoperation over the World Wide Web in particular, layer 3 provides a set of conventions for schema reference on the Web. Additional details on each of the three layers is provided in the sections below.

4.1 Layer 1: Summary of the Schema-validity Assessment Core

The fundamental purpose of the assessment core is to define assessment for a single element information item and its descendants with respect to a complex type definition. All processors are required to implement this core predicate in a manner which conforms exactly to this specification.

assessment is defined with reference to an XML Schema (note not a schema document) which consists of (at a minimum) the set of schema components (definitions and declarations) required for that assessment. This is not a circular definition, but rather a post facto observation: no element information item can be fully assessed unless all the components required by any aspect of its (potentially recursive) assessment are present in the schema.

As specified above, each schema component is associated directly or indirectly with a target namespace, or explicitly with no namespace. In the case of multi-namespace documents, components for more than one target namespace will co-exist in a schema.

Processors have the option to assemble (and perhaps to optimize or pre-compile) the entire schema prior to the start of an assessment episode, or to gather the schema lazily as individual components are required. In all cases it is required that:

- The processor succeed in locating the schema components transitively required to complete an assessment (note that components derived from schema documents can be integrated with components obtained through other means);
- no definition or declaration changes once it has been established;
- if the processor chooses to acquire declarations and definitions dynamically, that there be no side effects of such dynamic acquisition that would cause the results of assessment to differ from that which would have been obtained from the same schema components acquired in bulk.

NOTE: the assessment core is defined in terms of schema components at the abstract level, and no mention is made of the schema definition syntax (i.e. <schema>). Although many processors will acquire schemas in this format, others may operate on compiled representations, on a programmatic representation as exposed in some programming language, etc.

The obligation of a schema-aware processor as far as the assessment core is concerned is to implement one or more of the options for assessment, given below in Assessing Schema Validity (§5.2). Neither the choice of element information item for that assessment, nor which of the means of initiating assessment are used, is within the scope of this specification.
Although *assessment* is defined recursively, it is also intended to be implementable in streaming processors. Such processors may choose to incrementally assemble the schema during processing in response, for example, to encountering new namespaces. The implication of the invariants expressed above is that such incremental assembly must result in an *assessment* outcome that is the *same* as would be given if *assessment* was undertaken again with the final, fully assembled schema.

### 4.2 Layer 2: Schema Documents, Namespaces and Composition

#### 4.2.1 Assembling a schema for a single target namespace from multiple schema definition documents

Schema components for a single target namespace can be assembled from several *schema documents*, that is, several `<schema>` element information items:

XML Representation Summary: *include* Element Information Item

```
<include
  id = ID
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</include>
```

A `<schema>` information item may contain any number of `<include>` elements. Their `schemaLocation` attributes, consisting of a URI reference, identify other *schema documents*, that is `<schema>` information items.

The *XML Schemas*, corresponding to `<schema>` contains not only the components corresponding to its definition and declaration [children], but also all the components of all the *XML Schemas* corresponding to any `<include>`d schema documents. Such included schema documents must either (a) have the same
targetNamespace as the \texttt{<include>}ing schema document, or (b) no \texttt{targetNamespace} at all, in which case the \texttt{<include>}d schema document is converted to the \texttt{<include>}ing schema document's \texttt{targetNamespace}.

**Schema Representation Constraint: Inclusion Constraints and Semantics**

In addition to the conditions imposed on \texttt{<include>} element information items by the schema for schemas, all of the following must be true:

1. If the \texttt{.actual value} of the \texttt{schemaLocation} \texttt{[attribute]} successfully resolves one of the following must be true:
   1.1 It resolves to (a fragment of) a resource which is an XML document (of type \texttt{application/xml} or \texttt{text/xml} with an XML declaration for preference, but this is not required), which in turn corresponds to a \texttt{<schema>} element information item in a well-formed information set, which in turn corresponds to a valid schema.
   1.2 It resolves to a \texttt{<schema>} element information item in a well-formed information set, which in turn corresponds to a valid schema.

   In either case call the \texttt{<include>}d \texttt{<schema>} item \texttt{SII}, the valid schema \texttt{I} and the \texttt{<include>}ing item's parent \texttt{<schema>} item \texttt{SII}.

2. One of the following must be true:
   2.1 \texttt{SII} has a \texttt{targetNamespace} \texttt{[attribute]}, and its \texttt{.actual value} is identical to the \texttt{.actual value} of the \texttt{targetNamespace} \texttt{[attribute]} of \texttt{SII} (which must have such an \texttt{[attribute]}).
   2.2 Neither \texttt{SII} nor \texttt{SII} have a \texttt{targetNamespace} \texttt{[attribute]}.
   2.3 \texttt{SII} has no \texttt{targetNamespace} \texttt{[attribute]} (but \texttt{SII} does).

3. The appropriate case among the following must be true:
   3.1 If clause 2.1 or clause 2.2 above is satisfied, then the schema corresponding to \texttt{SII} must include not only definitions or declarations corresponding to the appropriate members of its own \texttt{[children]}, but also components identical to all the \texttt{.schema components} of \texttt{I}.
   3.2 If clause 2.3 above is satisfied, then the schema corresponding to the \texttt{<include>}d item's parent \texttt{<schema>} must include not only definitions or declarations corresponding to the appropriate members of its own \texttt{[children]}, but also components identical to all the \texttt{.schema components} of \texttt{I}, except that anywhere the \texttt{.absent} target namespace name would have appeared, the \texttt{.actual value} of the \texttt{targetNamespace} \texttt{[attribute]} of \texttt{SII} is used. In particular, it replaces \texttt{.absent}, in the following places:
      3.2.1 The \{target namespace\} of named schema components, both at the top level and (in the case of nested type definitions and nested attribute and element declarations whose \texttt{code} was \texttt{qualified}) nested within definitions;
      3.2.2 The \{namespace constraint\} of a wildcard, whether negated or not;

   It is not an error for the \texttt{.actual value} of the \texttt{schemaLocation} \texttt{[attribute]} to fail to resolve it all, in which case no corresponding inclusion is performed. It is an error for it to resolve but the rest of clause 1 above to fail to be satisfied. Failure to resolve may well cause less than complete \texttt{-assessment} outcomes, of course.

**NOTE:** As discussed in Missing Sub–components (§5.3), \texttt{-QName–s} in XML representations may fail to \texttt{-resolve–}, rendering components incomplete and unusable because of missing subcomponents. During schema construction, implementations are likely to retain \texttt{-QName–} values for such references, in case subsequent processing provides a referent. \texttt{-Absent} target \texttt{-namespace name–s} of such as–yet unresolved reference \texttt{-QName–s} in \texttt{<include>}d components should also be converted if clause 3.2 is satisfied.

**NOTE:** The above is carefully worded so that multiple \texttt{<include>}ing of the same schema document will not constitute a violation of clause 2 of Schema Properties Correct (§3.15.6), but applications are allowed, indeed encouraged, to avoid \texttt{<include>}ing the same schema document more than once to forestall the necessity of establishing identity component by component.

---

**4.2.2 Including modified component definitions**

In order to provide some support for evolution and versioning, it is possible to incorporate components corresponding to a schema document \textit{with modifications}. The modifications have a pervasive impact, that is, only the redefined components are used, even when referenced from other incorporated components, whether redefined themselves or not.
A `<schema>` information item may contain any number of `<redefine>` elements. Their `schemaLocation` attributes, consisting of a URI reference, identify other `<schema>` documents, that is `<schema>` information items.

The `<XML Schema>` corresponding to `<schema>` contains not only the components corresponding to its definition and declaration `<children>`, but also all the components of all the `<XML Schemas>` corresponding to any `<redefine>d schema documents`. Such schema documents must either (a) have the same `targetNamespace` as the `<redefine>`ing schema document, or (b) no `targetNamespace` at all, in which case the `<redefine>d schema document is converted to the `<redefine>`ing schema document's `targetNamespace`.

The definitions within the `<redefine>` element itself are restricted to be redefinitions of components from the `<redefine>d schema document, in terms of themselves`. That is,

- Type definitions must use themselves as their base type definition;
- Attribute group definitions and model group definitions must be supersets or subsets of their original definitions, either by including exactly one reference to themselves or by containing only (possibly restricted) components which appear in a corresponding way in their `<redefine>d selves`.

Not all the components of the `<redefine>d schema document need be redefined.

This mechanism is intended to provide a declarative and modular approach to schema modification, with functionality no different except in scope from what would be achieved by wholesale text copying and redefinition by editing. In particular redefining a type is not guaranteed to be side-effect free: it may have unexpected impacts on other type definitions which are based on the redefined one, even to the extent that some such definitions become ill-formed.

**NOTE:** The pervasive impact of redefinition reinforces the need for implementations to adopt some form of lazy or 'just-in-time' approach to component construction, which is also called for in order to avoid inappropriate dependencies on the order in which definitions and references appear in (collections of) schema documents.

Example

v1.xsd:
```xml
<xs:complexType name="personName">
  <xs:sequence>
    <xs:element name="title" minOccurs="0"/>
    <xs:element name="forename" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

```xml
<xs:element name="addressee" type="personName"/>
```
The schema corresponding to v2.xsd has everything specified by v1.xsd, with the personName type redefined, as well as everything it specifies itself. According to this schema, elements constrained by the personName type may end with a generation element. This includes not only the author element, but also the addressee element. **Schema Representation Constraint: Redefinition Constraints and Semantics**

In addition to the conditions imposed on `<redefine>` element information items by the schema for schemas all of the following must be true:1 If there are any element information items among the [children] other than `<annotation>` then the `actual value` of the schemaLocation [attribute] must successfully resolve.2 If the `actual value` of the schemaLocation [attribute] successfully resolves one of the following must be true:2.1 It resolves to (a fragment of) a resource which is an XML document (see clause 1.1), which in turn corresponds to a `<schema>` element information item in a well-formed information set, which in turn corresponds to a valid schema.2.2 It resolves to a `<schema>` element information item in a well-formed information set, which in turn corresponds to a valid schema. In either case call the `<redefine>d `<schema>` item SII, the valid schema I and the `<redefine>`ing item's parent `<schema>` item SII.3 One of the following must be true:3.1 SII has a targetNamespace [attribute], and its `actual value` is identical to the `actual value` of the targetNamespace [attribute] of SII (which must have such an [attribute]).3.2 Neither SII nor SII have a targetNamespace [attribute].3.3 SII has no targetNamespace [attribute] (but SII does).4 The appropriate case among the following must be true:4.1 If clause 3.1 or clause 3.2 above is satisfied, then the schema corresponding to SII must include not only definitions or declarations corresponding to the appropriate members of its own [children], but also components identical to all the ·schema components· of I, with the exception of those explicitly redefined (see Individual Component Redefinition (§4.2.2) below).4.2 If clause 3.3 above is satisfied, then the schema corresponding to SII must include not only definitions or declarations corresponding to the appropriate members of its own [children], but also components identical to all the ·schema components· of I, with the exception of those explicitly redefined (see Individual Component Redefinition (§4.2.2) below), except that anywhere the ·absent· target namespace name would have appeared, the `actual value` of the targetNamespace [attribute] of SII is used (see clause 3.2 in Inclusion Constraints and Semantics (§4.2.1) for details).5 Within the [children], each `<simpleType>` must have a `<restriction>` among its [children] and each `<complexType>` must have a restriction or extension among its grand-[children] the `actual value` of whose base [attribute] must be the same as the `actual value` of its own name attribute plus target namespace:6 Within the [children], for each `<group>` the appropriate case among the following must be true:6.1 If it has a `<group>` among its contents at some level the `actual value` of whose ref [attribute] is the same as the `actual value` of its own name attribute plus target namespace, then all of the following must be true:6.1.1 It must have exactly one such group.6.1.2 The `actual value` of both that group's minOccurs and maxOccurs [attribute] must be 1 (or ·absent·).6.2 If it has no such self-reference, then all of the following must be true:6.2.1 The `actual value` of its own name attribute plus target namespace must successfully resolve to a model group definition in I.6.2.2 The `[model group]` of the model group definition which corresponds to it per XML Representation of Model Group Definition
Schema Components (§3.7.2) must be a ·valid restriction· of the {model group} of that model group definition in I, as defined in Particle Valid (Restriction) (§3.9.6).7 Within the [children], for each <attributeGroup> the appropriate case among the following must be true:7.1 If it has an <attributeGroup> among its contents the ·actual value· of whose ref [attribute] is the same as the ·actual value· of its own name attribute plus target namespace, then it must have exactly one such group.7.2 If it has no such self-reference, then all of the following must be true:7.2.1 The ·actual value· of its own name attribute plus target namespace must successfully ·resolve· to an attribute group definition in I.7.2.2 The {attribute uses} and {attribute wildcard} of the attribute group definition which corresponds to it per XML Representation of Attribute Group Definition Schema Components (§3.6.2) must be ·valid restrictions· of the {attribute uses} and {attribute wildcard} of that attribute group definition in I, as defined in clause 2, clause 3 and clause 4 of Derivation Valid (Restriction, Complex) (§3.4.6) (where references to the base type definition are understood as references to the attribute group definition in I).

NOTE: An attribute group restrictively redefined per clause 7.2 corresponds to an attribute group whose {attribute uses} consist all and only of those attribute uses corresponding to <attribute>s explicitly present among the [children] of the <redefine>ing <attributeGroup>. No inheritance from the <redefine>d attribute group occurs. Its {attribute wildcard} is similarly based purely on an explicit <anyAttribute>, if present.

Schema Representation Constraint: Individual Component Redefinition
Corresponding to each non−<annotation> member of the [children] of a <redefine> there are one or two schema components in the <redefine>ing schema:1 The <simpleType> and <complexType> [children] information items each correspond to two components:1.1 One component which corresponds to the top−level definition item with the same name in the <redefine>d schema document, as defined in Schema Component Details (§3), except that its {name} is ·absent·;1.2 One component which corresponds to the information item itself, as defined in Schema Component Details (§3), except that its {base type definition} is the component defined in 1.1 above. This pairing ensures the coherence constraints on type definitions are respected, while at the same time achieving the desired effect, namely that references to names of redefined components in both the <redefine>ing and <redefine>d schema documents resolve to the redefined component as specified in 1.2 above.2 The <group> and <attributeGroup> [children] each correspond to a single component, as defined in Schema Component Details (§3), except that if and when a self-reference based on a ref [attribute] whose ·actual value· is the same as the item's name plus target namespace is resolved, a component which corresponds to the top−level definition item of that name and the appropriate kind in I is used. In all cases there must be a top−level definition item of the appropriate name and kind in the <redefine>d schema document.

NOTE: The above is carefully worded so that multiple equivalent <redefine>ing of the same schema document will not constitute a violation of clause 2 of Schema Properties Correct (§3.15.6), but applications are allowed, indeed encouraged, to avoid <redefine>ing the same schema document in the same way more than once to forestall the necessity of establishing identity component by component (although this will have to be done for the individual redefinitions themselves).

4.2.3 References to schema components across namespaces
As described in XML Schema Abstract Data Model (§2.2), every top−level schema component is associated with a target namespace (or, explicitly, with none). This section sets out the exact mechanism and syntax in the XML form of schema definition by which a reference to a foreign component is made, that is, a component with a different target namespace from that of the referring component.
Two things are required: not only a means of addressing such foreign components but also a signal to schema-aware processors that a schema document contains such references:

XML Representation Summary: import Element Information Item

```
<import
  id = ID
  namespace = anyURI
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}>  
Content:  (annotation?)  
</import>
```

The `<import>` element information item identifies namespaces used in external references, i.e. those whose QName identifies them as coming from a different namespace (or none) than the enclosing schema document's targetNamespace. The actual value of its namespace [attribute] indicates that the containing schema document may contain qualified references to schema components in that namespace (via one or more prefixes declared with namespace declarations in the normal way). If that attribute is absent, then the import allows unqualified reference to components with no target namespace. Note that components to be imported need not be in the form of a -schema document-; the processor is free to access or construct components using means of its own choosing.

The actual value of the schemaLocation, if present, gives a hint as to where a serialization of a -schema document-, with declarations and definitions for that namespace (or none) may be found. When no schemaLocation [attribute] is present, the schema author is leaving the identification of that schema to the instance, application or user, via the mechanisms described below in Layer 3: Schema Document Access and Web-interoperability (§4.3). When a schemaLocation is present, it must contain a single URI reference which the schema author warrants will resolve to a serialization of a -schema document- containing the component(s) in the `<import>`ed namespace referred to elsewhere in the containing schema document.

**NOTE:** Since both the namespace and schemaLocation [attribute] are optional, a bare `<import/>` information item is allowed. This simply allows unqualified reference to foreign components with no target namespace without giving any hints as to where to find them.

ExampleThe same namespace may be used both for real work, and in the course of defining schema components in terms of foreign components:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:html="http://www.w3.org/1999/xhtml"
       targetNamespace="uri:mywork" xmlns:my="uri:mywork">

  <import namespace="http://www.w3.org/1999/xhtml"/>

  <annotation>
    <documentation>
      <html:p>[Some documentation for my schema]</html:p>
    </documentation>
  </annotation>

  ...  

  <complexType name="myType">
```

4.2 Layer 2: Schema Documents, Namespaces and Composition
The treatment of references as QName implies that since (with the exception of the schema for schemas) the target namespace and the XML Schema namespace differ, without massive redeclaration of the default namespace either internal references to the names being defined in a schema document or the schema declaration and definition elements themselves must be explicitly qualified. This example takes the first option—most other examples in this specification have taken the second.

Schema Representation Constraint: Import Constraints and Semantics

In addition to the conditions imposed on <import> element information items by the schema for schemas all of the following must be true:

1. The appropriate case among the following must be true:
   1.1 If the namespace [attribute] is present, then its actual value must not match the actual value of the enclosing <schema>'s targetNamespace [attribute].
   1.2 If the namespace [attribute] is not present, then the enclosing <schema> must have a targetNamespace [attribute].

2. If the application schema reference strategy using the actual values of the schemaLocation and namespace [attributes], provides a referent, as defined by Schema Document Location Strategy (§4.3.2), one of the following must be true:
   2.1 The referent is (a fragment of) a resource which is an XML document (see clause 1.1), which in turn corresponds to a <schema> element information item in a well-formed information set, which in turn corresponds to a valid schema.
   2.2 The referent is a <schema> element information item in a well-formed information set, which in turn corresponds to a valid schema. In either case call the <schema> item SII and the valid schema I.

3. The appropriate case among the following must be true:
   3.1 If there is a namespace [attribute], then its actual value must be identical to the actual value of the targetNamespace [attribute] of SII.
   3.2 If there is no namespace [attribute], then SII must have no targetNamespace [attribute].

It is not an error for the application schema reference strategy to fail. It is an error for it to resolve but the rest of clause 2 above to fail to be satisfied. Failure to find a referent may well cause less than complete assessment outcomes, of course.

The _schema components_ (that is {type definitions}, {attribute declarations}, {element declarations}, {attribute group definitions}, {model group definitions}, {notation declarations}) of a schema corresponding to a <schema> element information item with one or more <import> element information items must include not only definitions or declarations corresponding to the appropriate members of its [children], but also, for each of those <import> element information items for which clause 2 above is satisfied, a set of _schema components_ identical to all the _schema components_ of I.

**NOTE:** The above is carefully worded so that multiple <import> ing of the same schema document will not constitute a violation of clause 2 of Schema Properties Correct (§3.15.6), but applications are allowed, indeed encouraged, to avoid <import> ing the same schema document more than once to forestall the necessity of establishing identity component by component. Given that the schemaLocation [attribute] is only a hint, it is open to applications to ignore all but the first <import> for a given namespace, regardless of the actual value of schemaLocation, but such a strategy risks missing useful information when new schemaLocations are offered.
4.3 Layer 3: Schema Document Access and Web–interoperability

4.3.1 Standards for representation of schemas and retrieval of schema documents on the Web

Layers 1 and 2 provide a framework for assessment and XML definition of schemas in a broad variety of environments. Over time, a range of standards and conventions may well evolve to support interoperability of XML Schema implementations on the World Wide Web. Layer 3 defines the minimum level of function required of all conformant processors operating on the Web: it is intended that, over time, future standards (e.g. XML Packages) for interoperability on the Web and in other environments can be introduced without the need to republish this specification.

4.3.1 Standards for representation of schemas and retrieval of schema documents on the Web

For interoperability, serialized schema documents, like all other Web resources, may be identified by URI and retrieved using the standard mechanisms of the Web (e.g. http, https, etc.) Such documents on the Web must be part of XML documents (see clause 1.1), and are represented in the standard XML schema definition form described by layer 2 (that is as <schema> element information items).

NOTE: there will often be times when a schema document will be a complete XML 1.0 document whose document element is <schema>. There will be other occasions in which <schema> items will be contained in other documents, perhaps referenced using fragment and/or XPointer notation.

NOTE: The variations among server software and web site administration policies make it difficult to recommend any particular approach to retrieval requests intended to retrieve serialized schema documents. An Accept header of application/xml, text/xml; q=0.9, */* is perhaps a reasonable starting point.

4.3.2 How schema definitions are located on the Web

As described in Layer 1: Summary of the Schema–validity Assessment Core (§4.1), processors are responsible for providing the schema components (definitions and declarations) needed for assessment. This section introduces a set of normative conventions to facilitate interoperability for instance and schema documents retrieved and processed from the Web.

NOTE: As discussed above in Layer 2: Schema Documents, Namespaces and Composition (§4.2), other non–Web mechanisms for delivering schemas for assessment may exist, but are outside the scope of this specification.

Processors on the Web are free to undertake assessment against arbitrary schemas in any of the ways set out in Assessing Schema–Validity (§5.2). However, it is useful to have a common convention for determining the schema to use. Accordingly, general–purpose schema–aware processors (i.e. those not specialized to one or a fixed set of pre–determined schemas) undertaking assessment of a document on the web must behave as follows:

• unless directed otherwise by the user, assessment is undertaken on the document element information item of the specified document;
• unless directed otherwise by the user, the processor is required to construct a schema corresponding to a schema document whose targetNamespace is identical to the namespace name, if any, of the
The composition of the complete schema for use in assessment is discussed in Layer 2: Schema Documents, Namespaces and Composition (§4.2) above. The means used to locate appropriate schema document(s) are processor and application dependent, subject to the following requirements:

1. Schemas are represented on the Web in the form specified above in Standards for representation of schemas and retrieval of schema documents on the Web (§4.3.1);
2. The author of a document uses namespace declarations to indicate the intended interpretation of names appearing therein; there may or may not be a schema retrievable via the namespace name. Accordingly whether a processor's default behavior is or is not to attempt such dereferencing, it must always provide for user-directed overriding of that default.

NOTE: Experience suggests that it is not in all cases safe or desirable from a performance point of view to dereference namespace names as a matter of course. User community and/or consumer/provider agreements may establish circumstances in which such dereference is a sensible default strategy; this specification allows but does not require particular communities to establish and implement such conventions. Users are always free to supply namespace names as schema location information when dereferencing is desired: see below.

3. On the other hand, in case a document author (human or not) created a document with a particular schema in view, and warrants that some or all of the document is conforms to that schema, the schemaLocation and noNamespaceSchemaLocation [attributes] (in the XML Schema instance namespace, that is, http://www.w3.org/2001/XMLSchema-instance) (hereafter xsi:schemaLocation and xsi:noNamespaceSchemaLocation) are provided. The first records the author's warrant with pairs of URI references (one for the namespace name, and one for a hint as to the location of a schema document defining names for that namespace name). The second similarly provides a URI reference as a hint as to the location of a schema document with no targetNamespace [attribute].

Unless directed otherwise, for example by the invoking application or by command line option, processors should attempt to dereference each schema document location URI in the actual value of such xsi:schemaLocation and xsi:noNamespaceSchemaLocation [attributes], see details below.

4. xsi:schemaLocation and xsi:noNamespaceSchemaLocation [attributes] can occur on any element. However, it is an error if such an attribute occurs after the first appearance of an element or attribute information item within an element information item initially validated, whose [namespace name] it addresses. According to the rules of Layer 1: Summary of the Schema-validity Assessment Core (§4.1), the corresponding schema may be lazily assembled, but is otherwise stable throughout assessment. Although schema location attributes can occur on any element, and can be processed incrementally as discovered, their effect is essentially global to the assessment. Definitions and declarations remain in effect beyond the scope of the element on which the binding is declared.

Example Multiple schema bindings can be declared using a single attribute. For example consider a stylesheet:

```xml
<stylesheet xmlns="http://www.w3.org/1999/XSL/Transform"
  xmlns:html="http://www.w3.org/1999/xhtml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xhtml="http://www.w3.org/1999/xhtml"/>
```
The namespace names used in `schemaLocation` can, but need not be identical to those actually qualifying the element within whose start tag it is found or its other attributes. For example, as above, all schema location information can be declared on the document element of a document, if desired, regardless of where the namespaces are actually used. **Schema Representation Constraint: Schema Document Location Strategy**

Given a namespace name (or none) and (optionally) a URI reference from `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation`, schema–aware processors may implement any combination of the following strategies, in any order: 1. Do nothing, for instance because a schema containing components for the given namespace name is already known to be available, or because it is known in advance that no efforts to locate schema documents will be successful (for example in embedded systems); 2. Based on the location URI, identify an existing schema document, either as a resource which is an XML document or a `<schema>` element information item, in some local schema repository; 3. Based on the namespace name, identify an existing schema document, either as a resource which is an XML document or a `<schema>` element information item, in some local schema repository; 4. Attempt to resolve the location URI, to locate a resource on the web which is or contains or references a `<schema>` element; 5. Attempt to resolve the namespace name to locate such a resource. Whenever possible configuration and/or invocation options for selecting and/or ordering the implemented strategies should be provided.

Improved or alternative conventions for Web interoperability can be standardized in the future without reopening this specification. For example, the W3C is currently considering initiatives to standardize the packaging of resources relating to particular documents and/or namespaces: this would be an addition to the mechanisms described here for layer 3. This architecture also facilitates innovation at layer 2: for example, it would be possible in the future to define an additional standard for the representation of schema components which allowed e.g. type definitions to be specified piece by piece, rather than all at once.

## 5 Schemas and Schema–validity Assessment

The architecture of schema–aware processing allows for a rich characterization of XML documents: schema validity is not a binary predicate.

This specification distinguishes between errors in schema construction and structure, on the one hand, and schema validation outcomes, on the other.

### 5.1 Errors in Schema Construction and Structure

Before _assessment_ can be attempted, a schema is required. Special–purpose applications are free to determine a schema for use in _assessment_, by whatever means are appropriate, but general purpose processors should implement the strategy set out in **Schema Document Location Strategy** (§4.3.2), starting with the namespaces declared in the document whose _assessment_ is being undertaken, and the _actual value_ of the `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` [attributes] thereof, if any, along with any other information about schema identity or schema document location provided by users in application–specific ways, if any.

It is an error if a schema and all the components which are the value of any of its properties, recursively, fail to satisfy all the relevant Constraints on Schemas set out in the last section of each of the subsections of **Schema Component Details** (§3).

If a schema is derived from one or more schema documents (that is, one or more `<schema>` element...
information items) based on the correspondence rules set out in Schema Component Details (§3) and Schemas and Namespaces: Access and Composition (§4), two additional conditions hold:

- It is an error if any such schema document would not be fully valid with respect to a schema corresponding to the Schema for Schemas (normative) (§A), that is, following schema-validation with such a schema, the <schema> element information items would have a [validation attempted] property with value full or partial and a [validity] property with value valid.
- It is an error if any such schema document is or contains any element information items which violate any of the relevant Schema Representation Constraints set out in Schema Representation Constraints (§C.3).

The three cases described above are the only types of error which this specification defines. With respect to the processes of the checking of schema structure and the construction of schemas corresponding to schema documents, this specification imposes no restrictions on processors after an error is detected. However -assessment with respect to schema–like entities which do not satisfy all the above conditions is incoherent. Accordingly, conformant processors must not attempt to undertake -assessment using such non–schemas.

5.2 Assessing Schema–Validity

With a schema which satisfies the conditions expressed in Errors in Schema Construction and Structure (§5.1) above, the schema–validity of an element information item can be assessed. Three primary approaches to this are possible:

1. The user or application identifies a complex type definition from among the {type definitions} of the schema, and appeals to Schema–Validity Assessment (Element) (§3.3.4) (clause 1.2); 2. The user or application identifies a element declaration from among the {element declarations} of the schema, checks that its {name} and {target namespace} match the [local name] and [namespace name] of the item, and appeals to Schema–Validity Assessment (Element) (§3.3.4) (clause 1.1); 3. The processor starts from Schema–Validity Assessment (Element) (§3.3.4) with no stipulated declaration or definition, and either -strict- or -lax- assessment ensues, depending on whether or not the element information and the schema determine either an element declaration (by name) or a type definition (via xsi:type) or not.

The outcome of this effort, in any case, will be manifest in the [validation attempted] and [validity] properties on the element information item and its [attributes] and [children], recursively, as defined by Assessment Outcome (Element) (§3.3.5) and Assessment Outcome (Attribute) (§3.2.5). It is up to applications to decide what constitutes a successful outcome.

Note that every element and attribute information item participating in the -assessment- will also have a [validation context] property which refers back to the element information item at which -assessment- began. [Definition:] This item, that is the element information item at which -assessment- began, is called the validation root.

NOTE: This specification does not reconstruct the XML 1.0 notion of root in either schemas or instances. Equivalent functionality is provided for at -assessment- invocation, via clause 2 above.

NOTE: This specification has nothing normative to say about multiple -assessment- episodes. It should however be clear from the above that if a processor restarts -assessment- with respect to a post–schema–validation infoset some post–schema–validation infoset contributions from the previous -assessment- may be overwritten. Restarting nonetheless may be useful, particularly at a node whose [validation attempted] property is none, in which case
there are three obvious cases in which additional useful information may result:

- **assessment**: was not attempted because of a **validation** failure, but declarations and/or definitions are available for at least some of the [children] or [attributes];
- **assessment**: was not attempted because a named definition or declaration was missing, but after further effort the processor has retrieved it.
- **assessment**: was not attempted because it was skipped, but the processor has at least some declarations and/or definitions available for at least some of the [children] or [attributes].

### 5.3 Missing Sub–components

At the beginning of Schema Component Details (§3), attention is drawn to the fact that most kinds of schema components have properties which are described therein as having other components, or sets of other components, as values, but that when components are constructed on the basis of their correspondence with element information items in schema documents, such properties usually correspond to QNames, and the **resolution** of such QNames may fail, resulting in one or more values of or containing **absent** where a component is mandated.

If at any time during **assessment**, an element or attribute information item is being **validated** with respect to a component of any kind any of whose properties has or contains such an **absent** value, the **validation** is modified, as following:

- In the case of attribute information items, the effect is as if clause 1 of Attribute Locally Valid (§3.2.4) had failed;
- In the case of element information items, the effect is as if clause 1 of Element Locally Valid (Element) (§3.3.4) had failed;
- In the case of element information items, processors may choose to continue **assessment**: see **lax assessment**.

Because of the value specification for [validation attempted] in Assessment Outcome (Element) (§3.3.5), if this situation ever arises, the document as a whole cannot show a [validation attempted] of full.

### 5.4 Responsibilities of Schema–aware Processors

Schema–aware processors are responsible for processing XML documents, schemas and schema documents, as appropriate given the level of conformance (as defined in Conformance (§2.4)) they support, consistently with the conditions set out above.

### A Schema for Schemas (normative)

The XML Schema definition for XML Schema: Structures itself is presented here as normative part of the specification, and as an illustrative example of the XML Schema in defining itself with the very constructs that it defines. The names of XML Schema language types, elements, attributes and groups defined here are evocative of their purpose, but are occasionally verbose.

There is some annotation in comments, but a fuller annotation will require the use of embedded documentation facilities or a hyperlinked external annotation for which tools are not yet readily available.
Since an XML Schema: Structures is an XML document, it has optional XML and doctype declarations that are provided here for completeness. The root schema element defines a new schema. Since this is a schema for XML Schema: Structures, the targetNamespace references the XML Schema namespace itself.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<!−− XML Schema schema for XML Schemas: Part 1: Structures −−>
<!DOCTYPE xs:schema PUBLIC "−//W3C//DTD XMLSCHEMA 200102//EN" "XMLSchema.dtd" [
<!−− provide ID type information even for parsers which only read the internal subset −−>
<!ATTLIST xs:schema          id  ID  #IMPLIED>
<!ATTLIST xs:complexType     id  ID  #IMPLIED>
<!ATTLIST xs:complexContent  id  ID  #IMPLIED>
<!ATTLIST xs:simpleContent   id  ID  #IMPLIED>
<!ATTLIST xs:extension       id  ID  #IMPLIED>
<!ATTLIST xs:element         id  ID  #IMPLIED>
<!ATTLIST xs:group           id  ID  #IMPLIED>
<!ATTLIST xs:all             id  ID  #IMPLIED>
<!ATTLIST xs:choice          id  ID  #IMPLIED>
<!ATTLIST xs:sequence        id  ID  #IMPLIED>
<!ATTLIST xs:any             id  ID  #IMPLIED>
<!ATTLIST xs:anyAttribute    id  ID  #IMPLIED>
<!ATTLIST xs:attribute       id  ID  #IMPLIED>
<!ATTLIST xs:attributeGroup  id  ID  #IMPLIED>
<!ATTLIST xs:unique          id  ID  #IMPLIED>
<!ATTLIST xs:key             id  ID  #IMPLIED>
<!ATTLIST xs:keyref          id  ID  #IMPLIED>
<!ATTLIST xs:selector        id  ID  #IMPLIED>
<!ATTLIST xs:field           id  ID  #IMPLIED>
<!ATTLIST xs:include         id  ID  #IMPLIED>
<!ATTLIST xs:import          id  ID  #IMPLIED>
<!ATTLIST xs:redefine        id  ID  #IMPLIED>
<!ATTLIST xs:notation        id  ID  #IMPLIED>
]>
<xs:annotation>
<xs:documentation source="http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/structures.html">
The schema corresponding to this document is normative, with respect to the syntactic constraints it expresses in the XML Schema language. The documentation (within &lt;documentation&gt; elements) below, is not normative, but rather highlights important aspects of the W3C Recommendation of which this is a part</xs:documentation>
</xs:annotation>

<xs:annotation>
<xs:documentation>
The simpleType element and all of its members are defined in datatypes.xsd</xs:documentation>
</xs:annotation>

<xs:include schemaLocation="datatypes.xsd"/>

<xs:annotation>
<xs:documentation>
Get access to the xml: attribute groups for xml:lang as declared on 'schema' and 'documentation' below
</xs:documentation>
</xs:annotation>
```
<xs:complexType name="openAttrs">
    <xs:annotation>
        <xs:documentation>
            This type is extended by almost all schema types to allow attributes from other namespaces to be added to user schemas.
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:restriction base="xs:anyType">
            <xs:anyAttribute namespace="##other" processContents="lax"/>
        </xs:restriction>
    </xs:complexContent>
</xs:complexType>

<xs:complexType name="annotated">
    <xs:annotation>
        <xs:documentation>
            This type is extended by all types which allow annotation other than &lt;schema&gt; itself
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="xs:openAttrs">
            <xs:sequence>
                <xs:element ref="xs:annotation" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:ID"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

<xs:group name="schemaTop">
    <xs:annotation>
        <xs:documentation>
            This group is for the elements which occur freely at the top level of schemas. All of their types are based on the "annotated" type by extension.
        </xs:documentation>
    </xs:annotation>
    <xs:choice>
        <xs:group ref="xs:redefinable"/>
        <xs:element ref="xs:element"/>  
        <xs:element ref="xs:attribute"/>
        <xs:element ref="xs:notation"/>
    </xs:choice>
</xs:group>

<xs:group name="redefinable">
    <xs:annotation>
        <xs:documentation>
            This group is for the elements which can self-redefine (see &lt;redefine> below).
        </xs:documentation>
    </xs:annotation>
    <xs:choice>
        <xs:element ref="xs:simpleType"/>
        <xs:element ref="xs:complexType"/>
        <xs:element ref="xs:group"/>
        <xs:element ref="xs:attributeGroup"/>
    </xs:choice>
</xs:group>
<xs:complexType>
  <xs:complexContent>
    <xs:extension base="xs:openAttrs">
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="xs:include"/>
        <xs:element ref="xs:import"/>
        <xs:element ref="xs:redefine"/>
        <xs:element ref="xs:annotation"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

5.3 Missing Sub–components
5.3 Missing Sub−components
<xs:attributeGroup name="occurs">
  <xs:annotation></xs:annotation>
  for all particles</xs:annotation>
  <xs:annotation>
    <xs:documentation>
      for all particles
    </xs:documentation>
  </xs:annotation>
  <xs:attribute name="minOccurs" type="xs:nonNegativeInteger" use="optional" default="1"/>
  <xs:attribute name="maxOccurs" type="xs:allNNI" use="optional" default="1"/>
</xs:attributeGroup>

<xs:attributeGroup name="defRef">
  <xs:annotation></xs:annotation>
  for element, group and attributeGroup,
  which both define and reference</xs:annotation>
  <xs:attribute name="name" type="xs:NCName"/>
  <xs:attribute name="ref" type="xs:QName"/>
</xs:attributeGroup>

<xs:group name="typeDefParticle">
  <xs:annotation>
    'complexType' uses this</xs:annotation>
  <xs:choice>
    <xs:element name="group" type="xs:groupRef"/>
    <xs:element ref="xs:all"/>
    <xs:element ref="xs:choice"/>
    <xs:element ref="xs:sequence"/>
  </xs:choice>
</xs:group>

<xs:group name="nestedParticle">
  <xs:choice>
    <xs:element name="element" type="xs:localElement"/>
    <xs:element name="group" type="xs:groupRef"/>
    <xs:element ref="xs:choice"/>
    <xs:element ref="xs:sequence"/>
    <xs:element ref="xs:any"/>
  </xs:choice>
</xs:group>

<xs:group name="particle">
  <xs:choice>
    <xs:element name="element" type="xs:localElement"/>
    <xs:element name="group" type="xs:groupRef"/>
    <xs:element ref="xs:all"/>
    <xs:element ref="xs:choice"/>
    <xs:element ref="xs:sequence"/>
    <xs:element ref="xs:any"/>
  </xs:choice>
</xs:group>

<xs:complexType name="attribute">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:sequence>
        <xs:element name="simpleType" minOccurs="0" type="xs:localSimpleType"/>
      </xs:sequence>
      <xs:attributeGroup ref="xs:defRef"/>
      <xs:attribute name="type" type="xs:QName"/>
      <xs:attribute name="use" use="optional" default="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

5.3 Missing Sub−components
<xs:restriction base="xs:NMTOKEN">
  <xs:enumeration value="prohibited"/>
  <xs:enumeration value="optional"/>
  <xs:enumeration value="required"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="default" type="xs:string"/>
<xs:attribute name="fixed" type="xs:string"/>
<xs:attribute name="form" type="xs:formChoice"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="topLevelAttribute">
<xs:complexContent>
<xs:restriction base="xs:attribute">
<xs:sequence>
  <xs:element ref="xs:annotation" minOccurs="0"/>
  <xs:element name="simpleType" minOccurs="0" type="xs:localSimpleType"/>
</xs:sequence>
<xs:attribute name="ref" use="prohibited"/>
<xs:attribute name="form" use="prohibited"/>
<xs:attribute name="use" use="prohibited"/>
<xs:attribute name="name" use="required" type="xs:NCName"/>
</xs:restriction>
</xs:complexContent>
</xs:complexType>
<xs:group name="attrDecls">
<xs:sequence>
<xs:choice minOccurs="0" maxOccurs="unbounded">
  <xs:element name="attribute" type="xs:attribute"/>
  <xs:element name="attributeGroup" type="xs:attributeGroupRef"/>
</xs:choice>
<xs:element ref="xs:anyAttribute" minOccurs="0"/>
</xs:sequence>
</xs:group>
<xs:element name="anyAttribute" type="xs:wildcard" id="anyAttribute">
<xs:annotation>
<xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-anyAttribute"/>
</xs:annotation>
</xs:element>
<xs:group name="complexTypeModel">
<xs:choice>
  <xs:element ref="xs:simpleContent"/>
  <xs:element ref="xs:complexContent"/>
  <xs:annotation>
    <xs:documentation>
      This branch is short for
    </xs:documentation>
    <xs:restriction base="xs:anyType">
      ...
    </xs:restriction>
  </xs:annotation>
</xs:choice>
</xs:group>

5.3 Missing Sub–components
5.3 Missing Sub–components
<xs:group ref="xs:simpleRestrictionModel" minOccurs="0"/>
</xs:choice>
<xs:group ref="xs:attrDecls"/>
</xs:sequence>
<xs:attribute name="base" type="xs:QName" use="required"/>
</xs:extension>
</xs:complexType>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="simpleRestrictionType">
<xs:complexContent>
<xs:restriction base="xs:restrictionType">
<xs:sequence>
<xs:element ref="xs:annotation" minOccurs="0"/>
<xs:group ref="xs:typeDefParticle" minOccurs="0"/>
<xs:group ref="xs:attrDecls"/>
</xs:sequence>
</xs:restriction>
</xs:complexContent>
</xs:complexType>

<xs:complexType name="extensionType">
<xs:complexContent>
<xs:extension base="xs:annotated">
<xs:choice>
<xs:element name="restriction" type="xs:complexRestrictionType"/>
<xs:element name="extension" type="xs:extensionType"/>
</xs:choice>
<xs:attribute name="mixed" type="xs:boolean">
<xs:annotation>
<xs:documentation>
Overrides any setting on complexType parent.
</xs:documentation>
</xs:annotation>
</xs:attribute>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="complexContent" id="complexContent">
<xs:annotation>
<xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element−complexContent"/>
</xs:annotation>
<xs:complexType>
<xs:complexContent>
<xs:extension base="xs:annotated">
<xs:choice>
<xs:element name="restriction" type="xs:complexRestrictionType"/>
<xs:element name="extension" type="xs:extensionType"/>
</xs:choice>
<xs:attribute name="mixed" type="xs:boolean">
<xs:annotation>
<xs:documentation>
Overrides any setting on complexType parent.
</xs:documentation>
</xs:annotation>
</xs:attribute>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>

<xs:complexType name="complexRestrictionType">
<xs:complexContent>
<xs:restriction base="xs:restrictionType">
<xs:sequence>
<xs:element ref="xs:annotation" minOccurs="0"/>
<xs:group ref="xs:typeDefParticle" minOccurs="0"/>
<xs:group ref="xs:attrDecls"/>
</xs:sequence>
</xs:restriction>
</xs:complexContent>
</xs:complexType>
</xs:element>

5.3 Missing Sub−components

XML Schema Part 0: Primer
5.3 Missing Sub–components
The element element can be used either at the top level to define an element-type binding globally, or within a content model to either reference a globally-defined element or type or declare an element-type binding locally. The ref form is not allowed at the top level.
<xs:complexType name="localElement">
  <xs:complexContent>
    <xs:restriction base="xs:element">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:choice minOccurs="0">
          <xs:element name="simpleType" type="xs:localSimpleType"/>
          <xs:element name="complexType" type="xs:localComplexType"/>
        </xs:choice>
        <xs:group ref="xs:identityConstraint" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="substitutionGroup" use="prohibited"/>
      <xs:attribute name="final" use="prohibited"/>
      <xs:attribute name="abstract" use="prohibited"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:element name="element" type="xs:topLevelElement" id="element">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-element"/>
  </xs:annotation>
</xs:element>

<xs:complexType name="group" abstract="true">
  <xs:annotation>
    <xs:documentation>
      group type for explicit groups, named top-level groups and
      group references
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:group ref="xs:particle" minOccurs="0" maxOccurs="unbounded"/>
      <xs:attributeGroup ref="xs:defRef"/>
      <xs:attributeGroup ref="xs:occurs"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="realGroup">
  <xs:complexContent>
    <xs:restriction base="xs:group">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:choice minOccurs="0" maxOccurs="1">
          <xs:element ref="xs:all"/>
          <xs:element ref="xs:choice"/>
          <xs:element ref="xs:sequence"/>
        </xs:choice>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="namedGroup">
  <xs:annotation>
    <xs:documentation>Should derive this from realGroup, but too complicated
    </xs:documentation>
  </xs:annotation>
</xs:complexType>

5.3 Missing Sub–components
for now</xs:documentation>
</xs:annotation>
<xs:sequence>
  <xs:element ref="xs:annotation" minOccurs="0" maxOccurs="0"/>
  <xs:choice minOccurs="1" maxOccurs="1">
    <xs:element name="all">
      <xs:complexType>
        <xs:complexContent>
          <xs:restriction base="xs:all">
            <xs:group ref="xs:allModel"/>
            <xs:attribute name="minOccurs" use="prohibited"/>
            <xs:attribute name="maxOccurs" use="prohibited"/>
          </xs:restriction>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
    <xs:element name="choice" type="xs:simpleExplicitGroup"/>
    <xs:element name="sequence" type="xs:simpleExplicitGroup"/>
  </xs:choice>
</xs:sequence>
<xs:attribute name="name" use="required" type="xs:NCName"/>
<xs:attribute name="ref" use="prohibited"/>
<xs:attribute name="minOccurs" use="prohibited"/>
<xs:attribute name="maxOccurs" use="prohibited"/>
</xs:complexType>
<xs:complexType name="groupRef">
  <xs:complexContent>
    <xs:restriction base="xs:realGroup">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0" maxOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="ref" use="required" type="xs:QName"/>
      <xs:attribute name="name" use="prohibited"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="explicitGroup">
  <xs:annotation>
    <xs:documentation>
group type for the three kinds of group</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:restriction base="xs:group">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0" maxOccurs="0"/>
        <xs:group ref="xs:nestedParticle" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="prohibited"/>
      <xs:attribute name="ref" type="xs:QName" use="prohibited"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="simpleExplicitGroup">
  <xs:complexContent>
    <xs:restriction base="xs:explicitGroup">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0" maxOccurs="0"/>
        <xs:group ref="xs:nestedParticle" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="all">
  <xs:annotation>
    <xs:documentation>
      Only elements allowed inside</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:restriction base="xs:explicitGroup">
      <xs:group ref="xs:allModel"/>
      <xs:attribute name="minOccurs" use="optional" default="1">
        <xs:simpleType>
          <xs:restriction base="xs:nonNegativeInteger">
            <xs:enumeration value="0"/>
            <xs:enumeration value="1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="maxOccurs" use="optional" default="1">
        <xs:simpleType>
          <xs:restriction base="xs:allNNI">
            <xs:enumeration value="0"/>
            <xs:enumeration value="1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
5.3 Missing Sub–components
simple type for the value of the 'namespace' attr of 'any' and 'anyAttribute'</xs:documentation>
</xs:annotation>
<xs:annotation>
<xs:documentation>
Value is
##any      − − any non-conflicting WFXML/attribute at all
##other    − − any non-conflicting WFXML/attribute from
namespace other than targetNS
##local    − − any unqualified non-conflicting WFXML/attribute
one or     − − any non-conflicting WFXML/attribute from
more URI        the listed namespaces
references (space separated)
##targetNamespace or ##local may appear in the above list, to
refer to the targetNamespace of the enclosing
schema or an absent targetNamespace respectively</xs:documentation>
</xs:annotation>
<xs:simpleType name="namespaceList">
<xs:annotation>
<xs:documentation>
A utility type, not for public use</xs:documentation>
</xs:annotation>
<xs:union>
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="##any"/>
<xs:enumeration value="##other"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType>
<xs:list>
<xs:union memberTypes="xs:anyURI">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="##targetNamespace"/>
<xs:enumeration value="##local"/>
</xs:restriction>
</xs:simpleType>
</xs:union>
</xs:simpleType>
<xs:simpleType>
</xs:list>
</xs:simpleType>
</xs:union>
</xs:simpleType>
</xs:element>

5.3 Missing Sub–components
<xs:element name="attribute" type="xs:topLevelAttribute" id="attribute">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-attribute"/>
  </xs:annotation>
</xs:element>

<xs:complexType name="attributeGroup" abstract="true">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:group ref="xs:attrDecls"/>
      <xs:attributeGroup ref="xs:defRef"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="namedAttributeGroup">
  <xs:complexContent>
    <xs:restriction base="xs:attributeGroup">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:attrDecls"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:NCName"/>
      <xs:attribute name="ref" use="prohibited"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="attributeGroupRef">
  <xs:complexContent>
    <xs:restriction base="xs:attributeGroup">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="ref" use="required" type="xs:QName"/>
      <xs:attribute name="name" use="prohibited"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:element name="attributeGroup" type="xs:namedAttributeGroup" id="attributeGroup">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-attributeGroup"/>
  </xs:annotation>
</xs:element>

<xs:element name="include" id="include">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-include"/>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:attribute name="schemaLocation" type="xs:anyURI" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="redefine" id="redefine">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-redefine"/>
  </xs:annotation>
</xs:element>

5.3 Missing Sub−components
5.3 Missing Sub–components
<xs:element name="field" id="field">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-field"/>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:attribute name="xpath" use="required">
          <xs:simpleType>
            <xs:annotation>
              <xs:documentation>A subset of XPath expressions for use in fields</xs:documentation>
              <xs:documentation>A utility type, not for public use</xs:documentation>
            </xs:annotation>
            <xs:restriction base="xs:token">
              <xs:annotation>
                <xs:documentation>The following pattern is intended to allow XPath expressions per the same EBNF as for selector, with the following change:
                Path ::= (\./)? ( Step '/' )* ( Step | '@' NameTest )
              </xs:annotation>
              <xs:pattern value="(\./)?\(((child::)?(\i\c*:)?(\i\c*|\*))\|(\i\c*|\*)\)*\(((child::)?(\i\c*:)?(\i\c*|\*))\|(\i\c*|\*)\)*">
              </xs:pattern>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:complexType name="keybase">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:sequence>
        <xs:element ref="xs:selector"/>
        <xs:element ref="xs:field" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:group name="identityConstraint">
  <xs:annotation>
    <xs:documentation>The three kinds of identity constraints, all with type of or derived from 'keybase'.
    </xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element ref="xs:unique"/>
    <xs:element ref="xs:key"/>
    <xs:element ref="xs:keyref"/>
  </xs:choice>
</xs:group>
5.3 Missing Sub–components
NOTE: And that is the end of the schema for XML Schema: Structures.

B References (normative)

XML 1.0 (Second Edition)
XML–Infoset

XML–Namespaces
Namespaces in XML, Tim Bray et al., eds., W3C, 14 January 1999. See http://www.w3.org/TR/1999/REC-xml-names-19990114/

XML Schema Requirements

XML Schemas: Datatypes

XPath

XPointer

C Outcome Tabulations (normative)

To facilitate consistent reporting of schema errors and -validation failures, this section tabulates and provides unique names for all the constraints listed in this document. Wherever such constraints have numbered parts, reports should use the name given below plus the part number, separated by a period (’.’). Thus for example cos-ct-extends.1.2 should be used to report a violation of the clause 1.2 of Derivation Valid (Extension) (§3.4.6).

C.1 Validation Rules

cvc–assess–attr
Schema–Validity Assessment (Attribute)
cvc–assess–elt
Schema–Validity Assessment (Element)
cvc–attribute
Attribute Locally Valid
cvc–au
Attribute Locally Valid (Use)
cvc–complex–type
Element Locally Valid (Complex Type)
cvc–datatype–valid
Datatype Valid
cvc–elt
Element Locally Valid (Element)
cvc–enumeration–valid
enumeration valid
cvc–facet–valid
Facet Valid
cvc–fractionDigits–valid
fractionDigits Valid
cvc–id
Validation Root Valid (ID/IDREF)
cvc−identity−constraint
   Identity−constraint Satisfied
cvc−length−valid
   Length Valid
cvc−maxExclusive−valid
   maxExclusive Valid
cvc−maxInclusive−valid
   maxInclusive Valid
cvc−maxLength−valid
   maxLength Valid
cvc−minExclusive−valid
   minExclusive Valid
cvc−minInclusive−valid
   minInclusive Valid
cvc−minLength−valid
   minLength Valid
cvc−model−group
   Element Sequence Valid
cvc−particle
   Element Sequence Locally Valid (Particle)
cvc−pattern−valid
   pattern valid
cvc−resolve−instance
   QName resolution (Instance)
cvc−simple−type
   String Valid
cvc−totalDigits−valid
   totalDigits Valid
cvc−type
   Element Locally Valid (Type)
cvc−wildcard
   Item Valid (Wildcard)
cvc−wildcard−namespace
   Wildcard allows Namespace Name

C.2 Contributions to the post–schema–validation infoset

attribute information item properties
   [attribute declaration] (Attribute Declaration)
   [member type definition] (Attribute Validated by Type)
   [member type definition anonymous] (Attribute Validated by Type)
   [member type definition name] (Attribute Validated by Type)
   [member type definition namespace] (Attribute Validated by Type)
   [schema default] (Attribute Validated by Type)
   [schema error code] (Validation Failure (Attribute))
   [schema normalized value] (Attribute Validated by Type)
   [schema specified] (Assessment Outcome (Attribute))
   [type definition] (Attribute Validated by Type)
   [type definition anonymous] (Attribute Validated by Type)
   [type definition name] (Attribute Validated by Type)
C.3 Schema Representation Constraints

schema_reference
  Schema Document Location Strategy
src−annotation
  Annotation Definition Representation OK
src−attribute
C.4 Schema Component Constraints
Annotation Correct

\textit{a-props-correct}

Attribute Declaration Properties Correct

\textit{au-props-correct}

Attribute Use Correct

\textit{c-fields-xpaths}

Fields Value OK

\textit{cos-all-limited}

All Group Limited

\textit{cos-applicable-facets}

applicable facets

\textit{cos-aw-intersect}

Attribute Wildcard Intersection

\textit{cos-aw-union}

Attribute Wildcard Union

\textit{cos-choice-range}

Effective Total Range (choice)

\textit{cos-ct-derived-ok}

Type Derivation OK (Complex)

\textit{cos-ct-extends}

Derivation Valid (Extension)

\textit{cos-element-consistent}

Element Declarations Consistent

\textit{cos-equiv-class}

Substitution Group

\textit{cos-equiv-derived-ok-rec}

Substitution Group OK (Transitive)

\textit{cos-group-emptiable}

Particle Emptiable

\textit{cos-list-of-atomic}

list of atomic

\textit{cos-no-circular-unions}

no circular unions

\textit{cos-nonambig}

Unique Particle Attribution

\textit{cos-ns-subset}

Wildcard Subset

\textit{cos-particle-extend}

Particle Valid (Extension)

\textit{cos-particle-restrict}

Particle Valid (Restriction)

\textit{cos-seq-range}

Effective Total Range (all and sequence)

\textit{cos-st-derived-ok}

Type Derivation OK (Simple)

\textit{cos-st-restricts}

Derivation Valid (Restriction, Simple)

\textit{cos-valid-default}

Element Default Valid (Immediate)

\textit{c-props-correct}

Identity-constraint Definition Properties Correct
c-selector-xpath
    Selector Value OK
c-t-props-correct
    Complex Type Definition Properties Correct
derivation-ok-restriction
    Derivation Valid (Restriction, Complex)
enumeration-required-notation
    enumeration facet value required for NOTATION
enumeration-valid-restriction
    enumeration valid restriction
e-props-correct
    Element Declaration Properties Correct
fractionDigits-totalDigits
    fractionDigits less than or equal to totalDigits
length-minLength-maxLength
    length and minLength or maxLength
length-valid-restriction
    length valid restriction
maxExclusive-valid-restriction
    maxExclusive valid restriction
maxInclusive-maxExclusive
    maxInclusive and maxExclusive
maxInclusive-valid-restriction
    maxInclusive valid restriction
maxLength-valid-restriction
    maxLength valid restriction
mg-props-correct
    Model Group Definition Properties Correct
mg-props-correct
    Model Group Correct
minExclusive-less-than-equal-to-maxExclusive
    minExclusive <= maxExclusive
minExclusive-less-than-maxInclusive
    minExclusive < maxInclusive
minExclusive-valid-restriction
    minExclusive valid restriction
minInclusive-less-than-equal-to-maxInclusive
    minInclusive <= maxInclusive
minInclusive-less-than-maxExclusive
    minInclusive < maxExclusive
minInclusive-minExclusive
    minInclusive and minExclusive
minInclusive-valid-restriction
    minInclusive valid restriction
minLength-less-than-equal-to-maxLength
    minLength <= maxLength
minLength-valid-restriction
    minLength valid restriction
no-xmlns
    xmlns Not Allowed
no-xsi
D Required Information Set Items and Properties (normative)

This specification requires as a precondition for assessment, an information set as defined in [XML-Infoset] which supports at least the following information items and properties:

**Attribute Information Item**
- [local name], [namespace name], [normalized value]

**Character Information Item**
- [character code]

**Element Information Item**
- [local name], [namespace name], [children], [attributes], [in-scope namespaces] or [namespace attributes]

**Namespace Information Item**
- [prefix], [namespace name]
This specification does not require any destructive alterations to the input information set: all the information set contributions specified herein are additive.

This appendix is intended to satisfy the requirements for Conformance to the [XML-Infoset] specification.

E Schema Components Diagram (non-normative)
The listing below is for the benefit of readers of a printed version of this document: it collects together all the definitions which appear in the document above.

**absent**
Throughout this specification, the term **absent** is used as a distinguished property value denoting absence.

**actual value**
The phrase **actual value** is used to refer to the member of the value space of the simple type definition associated with an attribute information item which corresponds to its -normalized value-

**assessment**
The word **assessment** is used to refer to the overall process of local validation, schema–validity assessment and infoset augmentation.

**base type definition**
A type definition used as the basis for an -extension- or -restriction- is known as the **base type definition** of that definition.

**component name**
Declarations and definitions may have and be identified by names, which are NCNames as defined by [XML–Namespaces]

**conformance to the XML Representation of Schemas**
-Minimally conforming- processors which accept schemas represented in the form of XML documents as described in Layer 2: Schema Documents, Namespaces and Composition (§4.2) are additionally said to provide **conformance to the XML Representation of Schemas**.

**content model**
A particle can be used in a complex type definition to constrain the -validation- of the [children] of an element information item; such a particle is called a **content model**

**context–determined declaration**
During -validation-, associations between element and attribute information items among the [children] and [attributes] on the one hand, and element and attribute declarations on the other, are established as a side–effect. Such declarations are called the **context–determined declarations**.

**declaration**
**declaration** components are associated by (qualified) name to information items being -validated-

**definition**
**definition** components define internal schema components that can be used in other schema components

**element substitution group**
Through the new mechanism of **element substitution groups**, XML Schemas provides a more powerful model supporting substitution of one named element for another

**extension**
A complex type definition which allows element or attribute content in addition to that allowed by another specified type definition is said to be an **extension**

**final**
The complex type is said to be **final**, because no further derivations are possible.

**fully conforming**
**Fully conforming** processors are network–enabled processors which are not only both -minimally conforming- and -in conformance to the XML Representation of Schemas-, but which additionally must be capable of accessing schema documents from the World Wide Web according to Representation of Schemas on the World Wide Web (§2.7) and How schema definitions are located on the Web (§4.3.2).
implicitly contains
A list of particles implicitly contains an element declaration if a member of the list contains that element declaration in its substitution group.

initial value
the initial value of some attribute information item is the value of the normalized value property of that item. Similarly, the initial value of an element information item is the string composed of, in order, the character code of each character information item in the children of that element information item.

item isomorphic to a component
by an item isomorphic to a component is meant an information item whose type is equivalent to the component’s, with one property per property of the component, with the same name, and value either the same atomic value, or an information item corresponding in the same way to its component value, recursively, as necessary.

laxly assessed
an element information item's schema validity may be laxly assessed if its context-determined declaration is not skip by validating with respect to the ur-type definition as per Element Locally Valid (Type) (§3.3.4).

minimally conforming
Minimally conforming processors must completely and correctly implement the Schema Component Constraints, Validation Rules, and Schema Information Set Contributions, contained in this specification.

NCName
An NCName is a name with no colon, as defined in XML Namespaces. When used in connection with the XML representation of schema components in this specification, this refers to the simple type NCName as defined in XML Schemas: Datatypes.

normalized value
The normalized value of an element or attribute information item is an initial value whose white space, if any, has been normalized according to the value of the whiteSpace facet of the simple type definition used in its validation:

partition
Define a partition of a sequence as a sequence of sub-sequences, some or all of which may be empty, such that concatenating all the sub-sequences yields the original sequence.

QName
A QName is a name with an optional namespace qualification, as defined in XML Namespaces. When used in connection with the XML representation of schema components or references to them, this refers to the simple type QName as defined in XML Schemas: Datatypes.

resolve
Whenever the word resolve in any form is used in this chapter in connection with a QName in a schema document, the following definition QName resolution (Schema Document) (§3.15.3) should be understood.

restriction
A type definition whose declarations or facets are in a one-to-one relation with those of another specified type definition, with each in turn restricting the possibilities of the one it corresponds to, is said to be a restriction.

schema component
Schema component is the generic term for the building blocks that comprise the abstract data model of the schema.

Schema Component Constraint
Constraints on the schema components themselves, i.e. conditions components must satisfy to be components at all. Located in the sixth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Component Constraints (§C.4).
A document in this form (i.e. a `<schema>` element information item) is a **schema document**.

**Schema Information Set Contribution**

Augmentations to post-schema-validation infosets expressed by schema components, which follow as a consequence of *validation* and/or *assessment*. Located in the fifth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Contributions to the post-schema-validation infoset (§C.2).

**Schema Representation Constraint**

Constraints on the representation of schema components in XML beyond those which are expressed in Schema for Schemas (normative) (§A). Located in the third sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Representation Constraints (§C.3).

**symbol space**

This specification introduces the term **symbol space** to denote a collection of names, each of which is unique with respect to the others.

**target namespace**

Several kinds of component have a **target namespace**, which is either *absent* or a namespace name, also as defined by [XML-Namespaces].

**type definition**

This specification uses the phrase **type definition** in cases where no distinction need be made between simple and complex types.

**Type Definition Hierarchy**

Except for a distinguished *ur-type definition*, every *type definition* is, by construction, either a *restriction* or an *extension* of some other type definition. The graph of these relationships forms a tree known as the **Type Definition Hierarchy**.

**ur-type definition**

A distinguished **ur-type definition** is present in each *XML Schema*, serving as the root of the type definition hierarchy for that schema.

**valid**

The word **valid** and its derivatives are used to refer to clause 1 above, the determination of local schema-validity.

**validation root**

This item, that is the element information item at which *assessment* began, is called the **validation root**.

**Validation Rules**

Contributions to *validation* associated with schema components. Located in the fourth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Validation Rules (§C.1).

**valid extension**

If this constraint Derivation Valid (Extension) (§3.4.6) holds of a complex type definition, it is a **valid extension** of its `{base type definition}`.

**valid restriction**

If this constraint Derivation Valid (Restriction, Complex) (§3.4.6) holds of a complex type definition, it is a **valid restriction** of its `{base type definition}`.

**valid restriction**

If this constraint Derivation Valid (Restriction, Simple) (§3.14.6) holds of a simple type definition, it is a **valid restriction** of its `{base type definition}`.

**XML Schema**

An **XML Schema** is a set of `schema components`. 

---

**F Glossary (non-normative)**
G DTD for Schemas (non–normative)

The DTD for XML Schema: Structures is given below. Note there is no implication here the schema must be the root element of a document.

Although this DTD is non–normative, any XML document which is not valid per this DTD, given redefinitions in its internal subset of the 'p' and 's' parameter entities below appropriate to its namespace declaration of the XML Schema namespace, is almost certainly not a valid schema document, with the exception of documents with multiple namespace prefixes for the XML Schema namespace itself. Accordingly authoring XML Schema documents using this DTD and DTD–based authoring tools, and specifying it as the DOCTYPE of documents intended to be XML Schema documents and validating them with a validating XML parser, are sensible development strategies which users are encouraged to adopt until XML Schema–based authoring tools and validators are more widely available.

<!-- DTD for XML Schemas: Part 1: Structures
Public Identifier: "−//W3C//DTD XMLSCHEMA 200102//EN"
Official Location: http://www.w3.org/2001/XMLSchema.dtd -->
<!−− Id: XMLSchema.dtd,v 1.30 2001/03/16 15:23:02 ht Exp −−>
<!−− With the exception of cases with multiple namespace
prefixes for the XML Schema namespace, any XML document which is not valid per this DTD given redefinitions in its internal subset of the
'p' and 's' parameter entities below appropriate to its namespace
declaration of the XML Schema namespace is almost certainly not
a valid schema. −−>

<!-- The simpleType element and its constituent parts
are defined in XML Schema: Part 2: Datatypes -->
<!ENTITY % xs−datatypes PUBLIC 'datatypes' 'datatypes.dtd'>
<!ENTITY % p 'xs:' > <!−− can be overridden in the internal subset of a
schema document to establish a different
namespace prefix −−>
<!ENTITY % s ':' xs'> <!−− if %p is defined (e.g. as foo:) then you must
also define %s as the suffix for the appropriate
namespace declaration (e.g. :foo) −−>
<!ENTITY % nds 'xmlns%s;'>

<!−− Define all the element names, with optional prefix −−>
<!ENTITY % schema "%p;schema">
<!ENTITY % complexType "%p;complexType">
<!ENTITY % complexContent "%p;complexContent">
<!ENTITY % simpleContent "%p;simpleContent">
<!ENTITY % extension "%p;extension">
<!ENTITY % element "%p;element">
<!ENTITY % unique "%p;unique">
<!ENTITY % key "%p;key">
<!ENTITY % keyref "%p;keyref">
<!ENTITY % selector "%p;selector">
<!ENTITY % field "%p;field">
<!ENTITY % group "%p;group">
<!ENTITY % all "%p;all">
<!ENTITY % choice "%p;choice">
<!ENTITY % sequence "%p;sequence">
<!ENTITY % any "%p;any">
<!ENTITY % anyAttribute "%p;anyAttribute">
<!ENTITY % attribute "%p;attribute">
<!ENTITY % attributeGroup "%p;attributeGroup">
<!ENTITY % include "%p;include"
Customisation entities for the ATTLIST of each element type.
   Define one of these if your schema takes advantage of the
   anyAttribute='##other' in the schema for schemas -->

<!ENTITY % schemaAttrs ''>
<!ENTITY % complexTypeAttrs ''>
<!ENTITY % complexContentAttrs ''>
<!ENTITY % simpleContentAttrs ''>
<!ENTITY % extensionAttrs ''>
<!ENTITY % elementAttrs ''>
<!ENTITY % groupAttrs ''>
<!ENTITY % allAttrs ''>
<!ENTITY % choiceAttrs ''>
<!ENTITY % sequenceAttrs ''>
<!ENTITY % anyAttrs ''>
<!ENTITY % anyAttributeAttrs ''>
<!ENTITY % attributeAttrs ''>
<!ENTITY % attributeGroupAttrs ''>
<!ENTITY % uniqueAttrs ''>
<!ENTITY % keyAttrs ''>
<!ENTITY % keyrefAttrs ''>
<!ENTITY % selectorAttrs ''>
<!ENTITY % fieldAttrs ''>
<!ENTITY % includeAttrs ''>
<!ENTITY % importAttrs ''>
<!ENTITY % redefineAttrs ''>
<!ENTITY % notationAttrs ''>
<!ENTITY % annotationAttrs ''>
<!ENTITY % appinfoAttrs ''>
<!ENTITY % documentationAttrs ''>

<!ENTITY % complexDerivationSet "CDATA">
   <!-- #all or space-separated list drawn from derivationChoice -->
<!ENTITY % blockSet "CDATA">
   <!-- #all or space-separated list drawn from
       derivationChoice + 'substitution' -->

<!ENTITY % mgs '"all; | %choice; | %sequence;">
<!ENTITY % cs '"choice; | %sequence;">
<!ENTITY % formValues '(qualified|unqualified)'>

<!ENTITY % attrDecls "((%attribute; | %attributeGroup;)*,(%anyAttribute;)?)'>
<!ENTITY % particleAndAttrs "((%mgs; | %group;)?, %attrDecls;)'>

!--- This is used in part2 -->
<!ENTITY % restriction1 "((%mgs; | %group;)?)'>

%xs-datatypes;

!--- the duplication below is to produce an unambiguous content model
which allows annotation everywhere -->

<!ELEMENT %schema; ((%include; | %import; | %redefine; | %annotation;)*,

((%simpleType; | %complexType;
  | %element; | %attribute;
  | %attributeGroup; | %group;
  | %notation; ),

(%%annotation();)* )>

<!ATTLIST %schema;
targetNamespace %URIref; #IMPLIED
version          CDATA                  #IMPLIED
%nds;            %URIref;               #FIXED 'http://www.w3.org/2001/XMLSchema'
xmlns            CDATA                  #IMPLIED
finalDefault     %complexDerivationSet; ''
blockDefault     %blockSet;            ''
id               ID                     #IMPLIED
elementFormDefault %formValues;       'unqualified'
attributeFormDefault %formValues;     'unqualified'
xmllang          CDATA                  #IMPLIED
%schemaAttrs;>

<!-- Note the xmlns declaration is NOT in the Schema for Schemas,
because at the Infoset level where schemas operate,
xmllns(:prefix) is NOT an attribute! -->

<!-- The declaration of xmlns is a convenience for schema authors -->

<!-- The id attribute here and below is for use in external references
from non-schemas using simple fragment identifiers.
It is NOT used for schema-to-schema reference, internal or
external. -->

<!-- a type is a named content type specification which allows attribute
declarations-->

<!-- -->

<!-- ELEMENT %complexType; (%annotation;?),
 (%simpleContent;|%complexType;|
  %particleAndAttrs;)}>

<!ATTLIST %complexType;
nname %NCName;                #IMPLIED
id    ID                    #IMPLIED
abstract %boolean;           #IMPLIED
final  %complexDerivationSet; #IMPLIED
block  %complexDerivationSet; #IMPLIED
mixed (true|false) 'false'
%complexTypeAttrs;>

<!-- particleAndAttrs is shorthand for a root type -->

<!-- mixed is disallowed if simpleContent, overridden if complexContent
has one too. -->

<!-- If anyAttribute appears in one or more referenced attributeGroups
and/or explicitly, the intersection of the permissions is used -->

<!-- ELEMENT %complexContent; (%restriction;|%extension;)

<!ATTLIST %complexContent;
mixed (true|false) #IMPLIED
id      ID                    #IMPLIED
%complexContentTypeAttrs;>

<!-- restriction should use the branch defined above, not the simple
one from part2; extension should use the full model -->
<!ELEMENT %simpleContent; (%restriction;|%extension;)>  
<!ATTLIST %simpleContent;  
  id ID #IMPLIED  
  %simpleContentAttrs;>  

<!−− restriction should use the simple branch from part2, not the one defined above; extension should have no particle −−>  

<!ELEMENT %extension; (%particleAndAttrs;)>  
<!ATTLIST %extension;  
  base %QName;      #REQUIRED  
  id ID #IMPLIED  
  %extensionAttrs;>  

<!−− an element is declared by either: a name and a type (either nested or referenced via the type attribute) or a ref to an existing element declaration −−>  

<!ELEMENT %element; ( (%annotation;)?, (%complexType; | %simpleType;)?,  
( %unique; | %key; | %keyref; )* ) >  
<!−− simpleType or complexType only if no type|ref attribute −−>  
<!−− ref not allowed at top level −−>  
<!ATTLIST %element;  
  name %NCName;               #IMPLIED  
  id ID #IMPLIED  
  ref %QName;                #IMPLIED  
  type %QName;                #IMPLIED  
  minOccurs %nonNegativeInteger; #IMPLIED  
  maxOccurs CDATA #IMPLIED  
  nillable %boolean; #IMPLIED  
  substitutionGroup %QName; #IMPLIED  
  abstract %boolean; #IMPLIED  
  final %complexDerivationSet; #IMPLIED  
  block %blockSet; #IMPLIED  
  default CDATA #IMPLIED  
  fixed CDATA #IMPLIED  
  form %formValues; #IMPLIED  
  %elementAttrs;>  

<!−− type and ref are mutually exclusive. name and ref are mutually exclusive, one is required −−>  
<!−− In the absence of type AND ref, type defaults to type of substitutionGroup, if any, else the ur-type, i.e. unconstrained −−>  
<!−− default and fixed are mutually exclusive −−>  

<!ELEMENT %group; ( (%annotation;)?, (%mgs;)?)>  
<!ATTLIST %group;  
  name %NCName;               #IMPLIED  
  ref %QName;                #IMPLIED  
  minOccurs %nonNegativeInteger; #IMPLIED  
  maxOccurs CDATA #IMPLIED  
  id ID #IMPLIED  
  %groupAttrs;>  

<!ELEMENT %all; ( (%annotation;)?, (%element;)* )>  
<!ATTLIST %all;  
  minOccurs (1) #IMPLIED  
  maxOccurs (1) #IMPLIED  
  id ID #IMPLIED  
  %allAttrs;>
XML Schema Part 0: Primer

<!ELEMENT %choice; (%annotation;)?, (%element; | %group; | %cs; | %any;)*>
<!ATTLIST %choice;
  minOccurs   %nonNegativeInteger;   #IMPLIED
  maxOccurs   CDATA                  #IMPLIED
  id          ID                     #IMPLIED
  %choiceAttrs;>

<!ELEMENT %sequence; (%annotation;)?, (%element; | %group; | %cs; | %any;)*>
<!ATTLIST %sequence;
  minOccurs   %nonNegativeInteger;   #IMPLIED
  maxOccurs   CDATA                  #IMPLIED
  id          ID                     #IMPLIED
  %sequenceAttrs;>

<!-- an anonymous grouping in a model, or
    a top-level named group definition, or a reference to same -->

<!-- Note that if order is 'all', group is not allowed inside.
    If order is 'all' THIS group must be alone (or referenced alone) at
    the top level of a content model -->

<!-- If order is 'all', minOccurs==maxOccurs==1 on element/any inside -->
<!-- Should allow minOccurs=0 inside order='all' . . . -->

<!ELEMENT %any; (%annotation;)?>
<!ATTLIST %any;
  namespace       CDATA                  '##any'
  processContents (skip|lax|strict)      'strict'
  minOccurs       %nonNegativeInteger;   '1'
  maxOccurs       CDATA                  '1'
  id              ID                     #IMPLIED
  %anyAttrs;>

<!-- namespace is interpreted as follows:
    ##any  -- any non-conflicting WFXML at all
    ##other -- any non-conflicting WFXML from namespace other
                than targetNamespace
    ##local  -- any unqualified non-conflicting WFXML/attribute
             one or more URI references
    ##targetNamespace ##local may appear in the above list,
                with the obvious meaning -->

<!ELEMENT %anyAttribute; (%annotation;)?>
<!ATTLIST %anyAttribute;
  namespace       CDATA              '##any'
  processContents (skip|lax|strict)  'strict'
  id              ID                 #IMPLIED
  %anyAttributeAttrs;>

<!-- namespace is interpreted as for 'any' above -->

<!-- simpleType only if no type|ref attribute -->
<!-- ref not allowed at top level, name iff at top level -->
<!ELEMENT %attribute; (%annotation;)?, (%simpleType;)?>
<!ATTLIST %attribute;
  name      %NCName;      #IMPLIED
  id        ID            #IMPLIED
  ref       %QName;       #IMPLIED

G DTD for Schemas (non-normative)  209
type QName; #IMPLIED
utype (prohibited|optional|required) #IMPLIED
default CDATA #IMPLIED
fixed CDATA #IMPLIED
form %formValues; #IMPLIED
%attributeAtts;
<!−− type and ref are mutually exclusive.
name and ref are mutually exclusive, one is required −−>
<!−− default for use is optional when nested, none otherwise −−>
<!−− default and fixed are mutually exclusive −−>
<!−− type attr and simpleType content are mutually exclusive −−>

<!−− an attributeGroup is a named collection of attribute decls, or a
reference thereto −−>
<!ELEMENT %attributeGroup; (%annotation;)?,
(attribute; | attributeGroup;)*,
(anyAttribute;)?) >
<!ATTLIST %attributeGroup;
name %NCName; #IMPLIED
id ID #IMPLIED
ref QName; #IMPLIED
%attributeGroupAttrs;>
<!−− ref iff no content, no name. ref iff not top level −−>
<!−− better reference mechanisms −−>
<!ELEMENT %unique; (%annotation;)?, %selector;, (%field;)+)>
<!ATTLIST %unique;
name %NCName; #REQUIRED
id ID #IMPLIED
%uniqueAttrs;>
<!ELEMENT %key; (%annotation;)?, %selector;, (%field;)+)>
<!ATTLIST %key;
name %NCName; #REQUIRED
id ID #IMPLIED
%keyAttrs;>
<!ELEMENT %keyref; (%annotation;)?, %selector;, (%field;)+)>
<!ATTLIST %keyref;
name %NCName; #REQUIRED
refer QName; #REQUIRED
id ID #IMPLIED
%keyrefAttrs;>
<!ELEMENT %selector; (%annotation;)?)>
<!ATTLIST %selector;
xpath %XPathExpr; #REQUIRED
id ID #IMPLIED
%selectorAttrs;>
<!ELEMENT %field; (%annotation;)?)>
<!ATTLIST %field;
xpath %XPathExpr; #REQUIRED
id ID #IMPLIED
%fieldAttrs;>

<!−− Schema combination mechanisms −−>
<!ELEMENT %include; (%annotation;)?)>
<!ATTLIST %include;
schemaLocation %URIref; #REQUIRED
id ID #IMPLIED
%includeAttrs;

<!ELEMENT %import; (%annotation;)?>
<!ATTLIST %import;
    namespace %URIref; #IMPLIED
    schemaLocation %URIref; #IMPLIED
    id ID      #IMPLIED
%importAttrs;>

<!ELEMENT %redefine; (%annotation; | %simpleType; | %complexType; |
    %attributeGroup; | %group;)*>
<!ATTLIST %redefine;
    schemaLocation %URIref; #REQUIRED
    id             ID       #IMPLIED
%redefineAttrs;>

<!ELEMENT %notation; (%annotation;)?>
<!ATTLIST %notation;
    name        %NCName;    #REQUIRED
    id          ID          #IMPLIED
    public      CDATA       #REQUIRED
    system      %URIref;    #IMPLIED
%notationAttrs;>

<!−− Annotation is either application information or documentation −−>
<!−− By having these here they are available for datatypes as well
as all the structures elements −−>

<!ELEMENT %annotation; (%appinfo; | %documentation;)*>
<!ATTLIST %annotation; %annotationAttrs;>
<!−− User must define annotation elements in internal subset for this
   to work −−>
<!ELEMENT %appinfo; ANY>   <!−− too restrictive −−>
<!ATTLIST %appinfo;
    source %URIref;      #IMPLIED
    id     ID             #IMPLIED
%appinfoAttrs;>

<!ELEMENT %documentation; ANY>   <!−− too restrictive −−>
<!ATTLIST %documentation;
    source %URIref;      #IMPLIED
    id     ID             #IMPLIED
xml:lang CDATA #IMPLIED
%documentationAttrs;>

<!NOTATION XMLSchemaStructures PUBLIC
'structures' 'http://www.w3.org/2001/XMLSchema.xsd' >
<!NOTATION XML PUBLIC
'REC−xml−1998−0210' 'http://www.w3.org/TR/1998/REC−xml−19980210' >

H Analysis of the Unique Particle Attribution Constraint
(non−normative)

A specification of the import of Unique Particle Attribution (§3.8.6) which does not appeal to a processing
model is difficult. What follows is intended as guidance, without claiming to be complete.

[Definition:]  Two non−group particles overlap if

H Analysis of the Unique Particle Attribution Constraint (non−normative) 211
They are both element declaration particles whose declarations have the same {name} and {target namespace}.

or

They are both element declaration particles one of which is in the other's ·substitution group·.

or

They are both wildcards, and the intensional intersection of their {namespace constraint}s as defined in Attribute Wildcard Intersection (§3.10.6) is not the empty set.

or

One is a wildcard and the other an element declaration, and the {target namespace} of the element declaration, or of any member of its ·substitution group·, is ·valid· with respect to the {namespace constraint} of the wildcard.

A content model will violate the unique attribution constraint if it contains two particles which ·overlap· and which either

• are both in the {particles} of a choice or all group

or

• may ·validate· adjacent information items and the first has {min occurs} less than {max occurs}.

Two particles may ·validate· adjacent information items if they are separated by at most epsilon transitions in the most obvious transcription of a content model into a finite−state automaton.

A precise formulation of this constraint can also be offered in terms of operations on finite−state automaton: transcribe the content model into an automaton in the usual way using epsilon transitions for optionality and unbounded maxOccurs, unfolding other numeric occurrence ranges and treating the heads of substitution groups as if they were choices over all elements in the group, but using not element QNames as transition labels, but rather pairs of element QNames and positions in the model. Determinize this automaton, treating wildcard transitions as opaque. Now replace all QName+position transition labels with the element QNames alone. If the result has any states with two or more identical−QName−labeled transitions from it, or a QName−labeled transition and a wildcard transition which subsumes it, or two wildcard transitions whose intentional intersection is non−empty, the model does not satisfy the Unique Attribution constraint.

I References (non−normative)

**DCD**

Document Content Description for XML (DCD), Tim Bray et al., eds., W3C, 10 August 1998. See http://www.w3.org/TR/1998/NOTE−dcd−19980731

**DDML**


**SOX**

Schema for Object−oriented XML, Andrew Davidson et al., eds., W3C, 1998. See
The following have contributed material to this draft:

- David Fallside, IBM
- Scott Lawrence, Agranat Systems
- Andrew Layman, Microsoft
- Eve L. Maler, Sun Microsystems
- Asir S. Vedamuthu, webMethods, Inc

The editors acknowledge the members of the XML Schema Working Group, the members of other W3C Working Groups, and industry experts in other forums who have contributed directly or indirectly to the process or content of creating this document. The Working Group is particularly grateful to Lotus Development Corp. and IBM for providing teleconferencing facilities.

The current members of the XML Schema Working Group are:

Jim Barnette, Defense Information Systems Agency (DISA); Paul V. Biron, Health Level Seven; Don Box, DevelopMentor; Allen Brown, Microsoft; Lee Buck, TIBCO Extensibility; Charles E. Campbell, Informix; Wayne Carr, Intel; Peter Chen, Bootstrap Alliance and LSU; David Cleary, Progress Software; Dan Connolly, W3C (staff contact); Ugo Corda, Xerox; Roger L. Costello, MITRE; Haavard Danielson, Progress Software; Josef Dietl, Mozquito Technologies; David Ezell, Hewlett–Packard Company; Alexander Falk, Altova GmbH; David Fallside, IBM; Dan Fox, Defense Logistics Information Service (DLIS); Matthew Fuchs, Commerce One; Andrew Goodchild, Distributed Systems Technology Centre (DSTC Pty Ltd); Paul Grosso, Arbortext, Inc; Martin Gudgin, DevelopMentor; Dave Hollander, Contivo, Inc (co–chair); Mary Holstege, Invited Expert; Jane Hunter, Distributed Systems Technology Centre (DSTC Pty Ltd); Rick Jelliffe, Academia Sinica; Simon Johnston, Rational Software; Bob Lojek, Mozquito Technologies; Ashok Malhotra, Microsoft; Lisa Martin, IBM; Noah Mendelsohn, Lotus Development Corporation; Adrian Michel, Commerce One; Alex Milowski, Invited Expert; Don Mullen, TIBCO Extensibility; Dave Peterson, Graphic Communications Association; Jonathan Robie, Software AG; Eric Sedlar, Oracle Corp.; C. M. Sperberg–McQueen, W3C (co–chair); Bob Streich, Calico Commerce; William K. Stumbo, Xerox; Henry S. Thompson, University of Edinburgh; Mark Tucker, Health Level Seven; Asir S. Vedamuthu, webMethods, Inc; Priscilla Walmsley, XMLSolutions; Norm Walsh, Sun Microsystems; Aki Yoshida, SAP AG; Kongyi Zhou, Oracle Corp.

The XML Schema Working Group has benefited in its work from the participation and contributions of a
number of people not currently members of the Working Group, including in particular those named below. Affiliations given are those current at the time of their work with the WG.

Paula Angerstein, Vignette Corporation; David Beech, Oracle Corp.; Gabe Beged-Dov, Rogue Wave Software; Greg Bumgardner, Rogue Wave Software; Dean Burson, Lotus Development Corporation; Mike Cokus, MITRE; Andrew Eisenberg, Progress Software; Rob Ellman, Calico Commerce; George Feinberg, Object Design; Charles Frankston, Microsoft; Ernesto Guerrieri, Inso; Michael Hyman, Microsoft; Renato Iannella, Distributed Systems Technology Centre (DSTC Pty Ltd); Dianne Kennedy, Graphic Communications Association; Janet Koenig, Sun Microsystems; Setrag Khoshafian, Technology Deployment International (TDI); Ara Kullukian, Technology Deployment International (TDI); Andrew Layman, Microsoft; Dmitry Lenkov, Hewlett-Packard Company; John McCarthy, Lawrence Berkeley National Laboratory; Murata Makoto, Xerox; Eve Maler, Sun Microsystems; Murray Maloney, Muzmo Communication, acting for Commerce One; Chris Olds, Wall Data; Frank Olken, Lawrence Berkeley National Laboratory; Shriram Revankar, Xerox; Mark Reinhold, Sun Microsystems; John C. Schneider, MITRE; Lew Shannon, NCR; William Shea, Merrill Lynch; Ralph Swick, W3C; Tony Stewart, Rivcom; Matt Timmermans, Microstar; Jim Trezzo, Oracle Corp.; Steph Tryphonas, Microstar code

W3C®
XML Schema Part 2: Datatypes

W3C Recommendation 02 May 2001

This version:
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/
(in XML and HTML, with a schema and DTD including datatype definitions, as well as a schema for built-in datatypes only, in a separate namespace.)

Latest version:
http://www.w3.org/TR/xmlschema-2/

Previous version:
http://www.w3.org/TR/2001/PR-xmlschema-2-20010330/

Editors:
Paul V. Biron (Kaiser Permanente, for Health Level Seven) <Paul.V.Biron@kp.org>
Ashok Malhotra (Microsoft, formerly of IBM) <ashokma@microsoft.com>

Copyright ©2001 W3C® (MIT, INRIA, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

Abstract

XML Schema: Datatypes is part 2 of the specification of the XML Schema language. It defines facilities for defining datatypes to be used in XML Schemas as well as other XML specifications. The datatype language, which is itself represented in XML 1.0, provides a superset of the capabilities found in XML 1.0 document type definitions (DTDs) for specifying datatypes on elements and attributes.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C’s role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document has been produced by the W3C XML Schema Working Group as part of the W3C XML Activity. The goals of the XML Schema language are discussed in the XML Schema Requirements document. The authors of this document are the XML Schema WG members. Different parts of this specification have different editors.

This version of this document incorporates some editorial changes from earlier versions.

Please report errors in this document to www-xml-schema-comments@w3.org (archive). The list of known errors in this specification is available at http://www.w3.org/2001/05/xmlschema-errata.

The English version of this specification is the only normative version. Information about translations of this document is available at http://www.w3.org/2001/05/xmlschema-translations.
A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR/.

**Table of contents**

1 Introduction  
1.1 Purpose  
1.2 Requirements  
1.3 Scope  
1.4 Terminology  
1.5 Constraints and Contributions  
2 Type System  
2.1 Datatype  
2.2 Value space  
2.3 Lexical space  
2.4 Facets  
2.5 Datatype dichotomies  
3 Built-in datatypes  
3.1 Namespace considerations  
3.2 Primitive datatypes  
3.3 Derived datatypes  
4 Datatype components  
4.1 Simple Type Definition  
4.2 Fundamental Facets  
4.3 Constraining Facets  
5 Conformance  

**Appendices**

A Schema for Datatype Definitions (normative)  
B DTD for Datatype Definitions (non-normative)  
C Datatypes and Facets  
D ISO 8601 Date and Time Formats  
E Adding durations to dateTimes  
F Regular Expressions  
G Glossary (non-normative)  
H References  
I Acknowledgements (non-normative)  

**1 Introduction**

1.1 Purpose

The [XML 1.0 (Second Edition)] specification defines limited facilities for applying datatypes to document content in that documents may contain or refer to DTDs that assign types to elements and attributes. However, document authors, including authors of traditional documents and those transporting data in XML, often require a higher degree of type checking to ensure robustness in document understanding and data interchange.

The table below offers two typical examples of XML instances in which datatypes are implicit: the instance on the left represents a billing invoice, the instance on the right a memo or perhaps an email message in XML.

<table>
<thead>
<tr>
<th>Data oriented</th>
<th>Document oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;invoice&gt;</code></td>
<td><code>&lt;memo importance='high' date='1999-03-23'&gt;</code></td>
</tr>
<tr>
<td><code>&lt;orderDate&gt;1999-01-21&lt;/orderDate&gt;</code></td>
<td><code>&lt;from&gt;Paul V. Biron&lt;/from&gt;</code></td>
</tr>
<tr>
<td><code>&lt;shipDate&gt;1999-01-25&lt;/shipDate&gt;</code></td>
<td><code>&lt;to&gt;Ashok Malhotra&lt;/to&gt;</code></td>
</tr>
<tr>
<td><code>&lt;billingAddress&gt;</code></td>
<td><code>&lt;subject&gt;Latest draft&lt;/subject&gt;</code></td>
</tr>
<tr>
<td><code>&lt;name&gt;Ashok Malhotra&lt;/name&gt;</code></td>
<td><code>&lt;body&gt;</code></td>
</tr>
<tr>
<td><code>&lt;street&gt;123 Microsoft Ave.&lt;/street&gt;</code></td>
<td><code>We need to discuss the latest draft &lt;emph&gt;immediately&lt;/emph&gt;.</code></td>
</tr>
<tr>
<td><code>&lt;city&gt;Hawthorne&lt;/city&gt;</code></td>
<td><code>Either email me at &lt;email&gt;</code></td>
</tr>
<tr>
<td><code>&lt;state&gt;NY&lt;/state&gt;</code></td>
<td><code>mailto:pa...</code></td>
</tr>
<tr>
<td><code>&lt;zip&gt;10532-0000&lt;/zip&gt;</code></td>
<td><code>or call &lt;phone&gt;555-9876&lt;/phone&gt;</code></td>
</tr>
<tr>
<td><code>&lt;billingAddress&gt;</code></td>
<td><code>&lt;/body&gt;</code></td>
</tr>
<tr>
<td><code>&lt;voice&gt;555-1234&lt;/voice&gt;</code></td>
<td><code>&lt;/memo&gt;</code></td>
</tr>
</tbody>
</table>
The invoice contains several dates and telephone numbers, the postal abbreviation for a state (which comes from an enumerated list of sanctioned values), and a ZIP code (which takes a definable regular form). The memo contains many of the same types of information: a date, telephone number, email address and an "importance" value (from an enumerated list, such as "low", "medium" or "high"). Applications which process invoices and memos need to raise exceptions if something that was supposed to be a date or telephone number does not conform to the rules for valid dates or telephone numbers.

In both cases, validity constraints exist on the content of the instances that are not expressible in XML DTDs. The limited datatyping facilities in XML have prevented validating XML processors from supplying the rigorous type checking required in these situations. The result has been that individual applications writers have had to implement type checking in an ad hoc manner. This specification addresses the need of both document authors and applications writers for a robust, extensible datatype system for XML which could be incorporated into XML processors. As discussed below, these datatypes could be used in other XML–related standards as well.

### 1.2 Requirements

The [XML Schema Requirements] document spells out concrete requirements to be fulfilled by this specification, which state that the XML Schema Language must:

1. provide for primitive data typing, including byte, date, integer, sequence, SQL and Java primitive datatypes, etc.;
2. define a type system that is adequate for import/export from database systems (e.g., relational, object, OLAP);
3. distinguish requirements relating to lexical data representation vs. those governing an underlying information set;
4. allow creation of user–defined datatypes, such as datatypes that are derived from existing datatypes and which may constrain certain of its properties (e.g., range, precision, length, format).

### 1.3 Scope

This portion of the XML Schema Language discusses datatypes that can be used in an XML Schema. These datatypes can be specified for element content that would be specified as #PCDATA and attribute values of various types in a DTD. It is the intention of this specification that it be usable outside of the context of XML Schemas for a wide range of other XML–related activities such as [XSL] and [RDF Schema].

### 1.4 Terminology

The terminology used to describe XML Schema Datatypes is defined in the body of this specification. The terms defined in the following list are used in building those definitions and in describing the actions of a datatype processor:

[Definition:] for compatibility

A feature of this specification included solely to ensure that schemas which use this feature remain compatible with [XML 1.0 (Second Edition)]

[Definition:] may

Conforming documents and processors are permitted to but need not behave as described.

[Definition:] match

(Of strings or names:) Two strings or names being compared must be identical. Characters with multiple possible representations in ISO/IEC 10646 (e.g. characters with both precomposed and
base+diacritic forms) match only if they have the same representation in both strings. No case folding is performed. (Of strings and rules in the grammar:) A string matches a grammatical production if it belongs to the language generated by that production.

[Definition:] must
Conforming documents and processors are required to behave as described; otherwise they are in error.

[Definition:] error
A violation of the rules of this specification; results are undefined. Conforming software may detect and report an error and may recover from it.

1.5 Constraints and Contributions

This specification provides three different kinds of normative statements about schema components, their representations in XML and their contribution to the schema−validation of information items:

[Definition:] Constraint on Schemas
Constraints on the schema components themselves, i.e. conditions components must satisfy to be components at all. Largely to be found in Datatype components (§4).

[Definition:] Schema Representation Constraint
Constraints on the representation of schema components in XML. Some but not all of these are expressed in Schema for Datatype Definitions (normative) (§A) and DTD for Datatype Definitions (non−normative) (§B).

[Definition:] Validation Rule
Constraints expressed by schema components which information items must satisfy to be schema−valid. Largely to be found in Datatype components (§4).

2 Type System

This section describes the conceptual framework behind the type system defined in this specification. The framework has been influenced by the [ISO 11404] standard on language−independent datatypes as well as the datatypes for [SQL] and for programming languages such as Java.

The datatypes discussed in this specification are computer representations of well known abstract concepts such as integer and date. It is not the place of this specification to define these abstract concepts; many other publications provide excellent definitions.

2.1 Datatype

[Definition:] In this specification, a datatype is a 3−tuple, consisting of a) a set of distinct values, called its value space, b) a set of lexical representations, called its lexical space, and c) a set of facet s that characterize properties of the value space, individual values or lexical items.

2.2 Value space

[Definition:] A value space is the set of values for a given datatype. Each value in the value space of a datatype is denoted by one or more literals in its lexical space.

The value space of a given datatype can be defined in one of the following ways:

• defined axiomatically from fundamental notions (intensional definition) [see primitive]
• enumerated outright (extensional definition) [see ·enumeration·]
• defined by restricting the ·value space· of an already defined datatype to a particular subset with a
given set of properties [see ·derived·]
• defined as a combination of values from one or more already defined ·value space·(s) by a specific
construction procedure [see ·list· and ·union·]

·value space·s have certain properties. For example, they always have the property of ·cardinality·, some
definition of equality and might be ·ordered·, by which individual values within the ·value space· can be
compared to one another. The properties of ·value space·s that are recognized by this specification are defined
in Fundamental facets (§2.4.1).

2.3 Lexical space

In addition to its ·value space·, each datatype also has a lexical space.

[Definition:] A lexical space is the set of valid literals for a datatype.

For example, "100" and "1.0E2" are two different literals from the ·lexical space· of float which both denote
the same value. The type system defined in this specification provides a mechanism for schema designers to
control the set of values and the corresponding set of acceptable literals of those values for a datatype.

NOTE: The literals in the ·lexical space·s defined in this specification have the following
characteristics:

Interoperability:
The number of literals for each value has been kept small; for many datatypes there is
a one–to–one mapping between literals and values. This makes it easy to exchange
the values between different systems. In many cases, conversion from
locale–dependent representations will be required on both the originator and the
recipient side, both for computer processing and for interaction with humans.

Basic readability:
Textual, rather than binary, literals are used. This makes hand editing, debugging, and
similar activities possible.

Ease of parsing and serializing:
Where possible, literals correspond to those found in common programming
languages and libraries.

2.3.1 Canonical Lexical Representation

While the datatypes defined in this specification have, for the most part, a single lexical representation i.e.
each value in the datatype's ·value space· is denoted by a single literal in its ·lexical space·, this is not always
the case. The example in the previous section showed two literals for the datatype float which denote the same
value. Similarly, there ·may· be several literals for one of the date or time datatypes that denote the same value
using different timezone indicators.

[Definition:] A canonical lexical representation is a set of literals from among the valid set of literals for
a datatype such that there is a one–to–one mapping between literals in the canonical lexical representation
and values in the ·value space·.
2.4 Facets

2.4.1 Fundamental facets

[Definition:] A **facet** is a single defining aspect of a _value space_. Generally speaking, each facet characterizes a _value space_ along independent axes or dimensions.

The facets of a datatype serve to distinguish those aspects of one datatype which _differ_ from other datatypes. Rather than being defined solely in terms of a prose description the datatypes in this specification are defined in terms of the _synthesis_ of facet values which together determine the _value space_ and properties of the datatype.

Facets are of two types: _fundamental_ facets that define the datatype and _non–fundamental or constraining_ facets that constrain the permitted values of a datatype.

2.4.1 Fundamental facets

[Definition:] A **fundamental facet** is an abstract property which serves to semantically characterize the values in a _value space_.

All **fundamental facets** are fully described in Fundamental Facets (§4.2).

2.4.2 Constraining or Non–fundamental facets

[Definition:] A **constraining facet** is an optional property that can be applied to a datatype to constrain its _value space_.

Constraining the _value space_ consequently constrains the _lexical space_. Adding _constraining facet_ s to a _base type_ is described in Derivation by restriction (§4.1.2.1).

All **constraining facets** are fully described in Constraining Facets (§4.3).

2.5 Datatype dichotomies

2.5.1 Atomic vs. list vs. union datatypes

2.5.2 Primitive vs. derived datatypes

2.5.3 Built–in vs. user–derived datatypes

It is useful to categorize the datatypes defined in this specification along various dimensions, forming a set of characterization dichotomies.

2.5.1 Atomic vs. list vs. union datatypes

The first distinction to be made is that between _atomic_, _list_, and _union_ datatypes.

- [Definition:] **Atomic** datatypes are those having values which are regarded by this specification as being indivisible.
- [Definition:] **List** datatypes are those having values each of which consists of a finite–length (possibly empty) sequence of values of an _atomic_ datatype.
- [Definition:] **Union** datatypes are those whose _value space_ s and _lexical space_ s are the union of the _value space_ s and _lexical space_ s of one or more other datatypes.
For example, a single token which matches Nmtoken from [XML 1.0 (Second Edition)] could be the value of an atomic datatype (NM_TOKEN); while a sequence of such tokens could be the value of a list datatype (NM_TOKENS).

2.5.1.1 Atomic datatypes

Atomic datatypes can be either primitive or derived. The value space of an atomic datatype is a set of "atomic" values, which for the purposes of this specification, are not further decomposable. The lexical space of an atomic datatype is a set of literals whose internal structure is specific to the datatype in question.

2.5.1.2 List datatypes

Several type systems (such as the one described in [ISO 11404]) treat list datatypes as special cases of the more general notions of aggregate or collection datatypes.

List datatypes are always derived. The value space of a list datatype is a set of finite-length sequences of atomic values. The lexical space of a list datatype is a set of literals whose internal structure is a white space separated sequence of literals of the atomic datatype of the items in the list (where whitespace matches S in [XML 1.0 (Second Edition)]).

[Definition:] The atomic datatype that participates in the definition of a list datatype is known as the itemType of that list datatype.

Example

```xml
<simpleType name='sizes'>
  <list itemType='decimal'/>
</simpleType>

<cerealSizes xsi:type='sizes'> 8 10.5 12 </cerealSizes>
```

A list datatype can be derived from an atomic datatype whose lexical space allows whitespace (such as string or anyURI). In such a case, regardless of the input, list items will be separated at whitespace boundaries.

Example

```xml
<simpleType name='listOfString'>
  <list itemType='string'/>
</simpleType>

<someElement xsi:type='listOfString'>
  this is not list item 1
  this is not list item 2
  this is not list item 3
</someElement>
```

In the above example, the value of the someElement element is not a list of length 3; rather, it is a list of length 18.

When a datatype is derived from a list datatype, the following constraining facets apply:

- length.
For each of \textit{length}, \textit{maxLength}, and \textit{minLength}, the \textit{unit of length} is measured in number of list items. The value of \textit{whiteSpace} is fixed to the value \textit{collapse}.

The \textit{canonical-lexical-representation} for the \textit{list} datatype is defined as the lexical form in which each item in the \textit{list} has the canonical lexical representation of its \textit{itemType}.

\subsection{2.5.1.3 Union datatypes}

The \textit{value space} and \textit{lexical space} of a \textit{union} datatype are the union of the \textit{value space}s and \textit{lexical space}s of its \textit{memberTypes}. \textit{Union} datatypes are always \textit{derived}. Currently, there are no \textit{built-in} \textit{union} datatypes.

Example A prototypical example of a \textit{union} type is the \textit{maxOccurs} attribute on the \textit{element} element in XML Schema itself: it is a union of nonNegativeInteger and an enumeration with the single member, the string "unbounded", as shown below.

```xml
<attributeGroup name="occurs">
  <attribute name="minOccurs" type="nonNegativeInteger" default="1"/>
  <attribute name="maxOccurs">
    <simpleType>
      <union>
        <simpleType>
          <restriction base='nonNegativeInteger'/>
        </simpleType>
        <simpleType>
          <restriction base='string'>
            <enumeration value='unbounded'/>
          </restriction>
        </simpleType>
      </union>
    </simpleType>
  </attribute>
</attributeGroup>
```

Any number (greater than 1) of \textit{atomic} or \textit{list} \textit{datatype}s can participate in a \textit{union} type.

[Definition:] The datatypes that participate in the definition of a \textit{union} datatype are known as the \textit{memberTypes} of that \textit{union} datatype.

The order in which the \textit{memberTypes} are specified in the definition (that is, the order of the \textit{simpleType} children of the \textit{union} element, or the order of the QNames in the \textit{memberTypes} attribute) is significant. During validation, an element or attribute's value is validated against the \textit{memberTypes} in the order in which they appear in the definition until a match is found. The evaluation order can be overridden with the use of \textit{xsi:type}.

Example For example, given the definition below, the first instance of the \textit{size} element validates correctly as an \textit{integer} (§3.3.13), the second and third as \textit{string} (§3.2.1).
The canonical-lexical-representation for a _union_ datatype is defined as the lexical form in which the values have the canonical lexical representation of the appropriate _memberTypes_.

NOTE: A datatype which is _atomic_ in this specification need not be an "atomic" datatype in any programming language used to implement this specification. Likewise, a datatype which is a _list_ in this specification need not be a "list" datatype in any programming language used to implement this specification. Furthermore, a datatype which is a _union_ in this specification need not be a "union" datatype in any programming language used to implement this specification.

### 2.5.2 Primitive vs. derived datatypes

Next, we distinguish between _primitive_ and _derived_ datatypes.

- [Definition:] **Primitive** datatypes are those that are not defined in terms of other datatypes; they exist _ab initio_.
- [Definition:] **Derived** datatypes are those that are defined in terms of other datatypes.

For example, in this specification, `float` is a well-defined mathematical concept that cannot be defined in terms of other datatypes, while a `integer` is a special case of the more general datatype `decimal`.

[Definition:] There exists a conceptual datatype, whose name is `anySimpleType`, that is the simple version of the _ur-type definition_ from [XML Schema Part 1: Structures]. `anySimpleType` can be considered as the _base type_ of all _primitive_ types. The _value space_ of `anySimpleType` can be considered to be the _union_ of the _value space_ of all _primitive_ datatypes.

The datatypes defined by this specification fall into both the _primitive_ and _derived_ categories. It is felt that a judiciously chosen set of _primitive_ datatypes will serve the widest possible audience by providing a set of convenient datatypes that can be used as is, as well as providing a rich enough base from which the variety of datatypes needed by schema designers can be _derived_.

In the example above, `integer` is _derived_ from `decimal`.

NOTE: A datatype which is _primitive_ in this specification need not be a "primitive" datatype in any programming language used to implement this specification. Likewise, a datatype which is _derived_ in this specification need not be a "derived" datatype in any
programming language used to implement this specification.

As described in more detail in XML Representation of Simple Type Definition Schema Components (§4.1.2), each user-derived datatype must be defined in terms of another datatype in one of three ways: 1) by assigning constraining facets which serve to restrict the value space of the user-derived datatype to a subset of that of the base type; 2) by creating a list datatype whose value space consists of finite-length sequences of values of its itemType; or 3) by creating a union datatype whose value space consists of the union of the value space its memberTypes.

2.5.2.1 Derived by restriction

[Definition:] A datatype is said to be derived by restriction from another datatype when values for zero or more constraining facets are specified that serve to constrain its value space; and/or its lexical space; to a subset of those of its base type.

[Definition:] Every datatype that is derived by restriction is defined in terms of an existing datatype, referred to as its base type. base types can be either primitive or derived.

2.5.2.2 Derived by list

A list datatype can be derived from another datatype (its itemType) by creating a value space that consists of a finite-length sequence of values of its itemType.

2.5.2.3 Derived by union

One datatype can be derived from one or more datatypes by unioning their value spaces and, consequently, their lexical spaces.

2.5.3 Built-in vs. user-derived datatypes

• [Definition:] Built-in datatypes are those which are defined in this specification, and can be either primitive or derived;
• [Definition:] User-derived datatypes are those derived datatypes that are defined by individual schema designers.

Conceptually there is no difference between the built-in, derived datatypes included in this specification and the user-derived, datatypes which will be created by individual schema designers. The built-in, derived datatypes are those which are believed to be so common that if they were not defined in this specification many schema designers would end up "reinventing" them. Furthermore, including these derived datatypes in this specification serves to demonstrate the mechanics and utility of the datatype generation facilities of this specification.

NOTE: A datatype which is built-in in this specification need not be a "built-in" datatype in any programming language used to implement this specification. Likewise, a datatype which is user-derived in this specification need not be a "user-derived" datatype in any programming language used to implement this specification.

3 Built-in datatypes

2.5 Datatype dichotomies
Each built-in datatype in this specification (both primitive and derived) can be uniquely addressed via a URI Reference constructed as follows:

1. the base URI is the URI of the XML Schema namespace
2. the fragment identifier is the name of the datatype

For example, to address the int datatype, the URI is:

- `http://www.w3.org/2001/XMLSchema#int`

Additionally, each facet definition element can be uniquely addressed via a URI constructed as follows:
1. the base URI is the URI of the XML Schema namespace
2. the fragment identifier is the name of the facet

For example, to address the maxInclusive facet, the URI is:

- http://www.w3.org/2001/XMLSchema#maxInclusive

Additionally, each facet usage in a built-in datatype definition can be uniquely addressed via a URI constructed as follows:

1. the base URI is the URI of the XML Schema namespace
2. the fragment identifier is the name of the datatype, followed by a period (".") followed by the name of the facet

For example, to address the usage of the maxInclusive facet in the definition of int, the URI is:

- http://www.w3.org/2001/XMLSchema#int.maxInclusive

### 3.1 Namespace considerations

The built-in datatypes defined by this specification are designed to be used with the XML Schema definition language as well as other XML specifications. To facilitate usage within the XML Schema definition language, the built-in datatypes in this specification have the namespace name:

- http://www.w3.org/2001/XMLSchema

To facilitate usage in specifications other than the XML Schema definition language, such as those that do not want to know anything about aspects of the XML Schema definition language other than the datatypes, each built-in datatype is also defined in the namespace whose URI is:

- http://www.w3.org/2001/XMLSchema-datatypes

This applies to both built-in, primitive- and built-in, derived- datatypes.

Each user-derived datatype is also associated with a unique namespace. However, user-derived datatypes do not come from the namespace defined by this specification; rather, they come from the namespace of the schema in which they are defined (see XML Representation of Schemas in [XML Schema Part 1: Structures]).

### 3.2 Primitive datatypes

3.2.1 string3.2.2 boolean3.2.3 decimal3.2.4 float3.2.5 double3.2.6 duration3.2.7 dateTime3.2.8 time3.2.9 date3.2.10 gYearMonth3.2.11 gYear3.2.12 gMonthDay3.2.13 gDay3.2.14 gMonth3.2.15 hexBinary3.2.16 base64Binary3.2.17 anyURI3.2.18 QName3.2.19 NOTATION

The primitive datatypes defined by this specification are described below. For each datatype, the value space and lexical space are defined, constraining facet-s which apply to the datatype are listed and any datatypes derived from this datatype are specified.

-primitive datatypes can only be added by revisions to this specification.
3.2.1 string

[Definition:] The string datatype represents character strings in XML. The value space of string is the set of finite-length sequences of characters (as defined in [XML 1.0 (Second Edition)]) that match the Char production from [XML 1.0 (Second Edition)]. A character is an atomic unit of communication; it is not further specified except to note that every character has a corresponding Universal Character Set code point, which is an integer.

NOTE: Many human languages have writing systems that require child elements for control of aspects such as bidirectional formatting or ruby annotation (see [Ruby] and Section 8.2.4 Overriding the bidirectional algorithm: the BDO element of [HTML 4.01]). Thus, string, as a simple type that can contain only characters but not child elements, is often not suitable for representing text. In such situations, a complex type that allows mixed content should be considered. For more information, see Section 5.5 Any Element, Any Attribute of [XML Schema Language: Part 2 Primer].

NOTE: As noted in ordered, the fact that this specification does not specify an order-relation for string does not preclude other applications from treating strings as being ordered.

3.2.1.1 Constraining facets

string has the following constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

3.2.1.2 Derived datatypes

The following built-in datatypes are derived from string:

- normalizedString

3.2.2 boolean

[Definition:] boolean has the value space required to support the mathematical concept of binary-valued logic: {true, false}.

3.2.2.1 Lexical representation

An instance of a datatype that is defined as boolean can have the following legal literals {true, false, 1, 0}.

3.2.2.2 Canonical representation

The canonical representation for boolean is the set of literals {true, false}. 
3.2.2.3 Constraining facets

`boolean` has the following constraining facets:

- `pattern`
- `whiteSpace`

3.2.3 decimal

[Definition:] `decimal` represents arbitrary precision decimal numbers. The value space of `decimal` is the set of the values $i \times 10^{n}$, where $i$ and $n$ are integers such that $n \geq 0$. The order relation on `decimal` is: $x < y$ iff $y - x$ is positive.

[Definition:] The value space of types derived from `decimal` with a value for `totalDigits` of $p$ is the set of values $i \times 10^{n}$, where $n$ and $i$ are integers such that $p \geq n \geq 0$ and the number of significant decimal digits in $i$ is less than or equal to $p$.

[Definition:] The value space of types derived from `decimal` with a value for `fractionDigits` of $s$ is the set of values $i \times 10^{n}$, where $i$ and $n$ are integers such that $0 \leq n \leq s$.

**NOTE:** All minimally conforming processors must support decimal numbers with a minimum of 18 decimal digits (i.e., with a `totalDigits` of 18). However, minimally conforming processors may set an application-defined limit on the maximum number of decimal digits they are prepared to support, in which case that application-defined maximum number must be clearly documented.

3.2.3.1 Lexical representation

`decimal` has a lexical representation consisting of a finite-length sequence of decimal digits (#x30–#x39) separated by a period as a decimal indicator. If `totalDigits` is specified, the number of digits must be less than or equal to `totalDigits`. If `fractionDigits` is specified, the number of digits following the decimal point must be less than or equal to the `fractionDigits`. An optional leading sign is allowed. If the sign is omitted, "+" is assumed. Leading and trailing zeroes are optional. If the fractional part is zero, the period and following zero(es) can be omitted. For example: `-1.23, 12678967.543233, +100000.00, 210`.

3.2.3.2 Canonical representation

The canonical representation for `decimal` is defined by prohibiting certain options from the Lexical representation (§3.2.3.1). Specifically, the preceding optional "+" sign is prohibited. The decimal point is required. Leading and trailing zeroes are prohibited subject to the following: there must be at least one digit to the right and to the left of the decimal point which may be a zero.

3.2.3.3 Constraining facets

`decimal` has the following constraining facets:

- `totalDigits`
- `fractionDigits`
- `pattern`
- `whiteSpace`
- `enumeration`

3.2 Primitive datatypes 228
3.2 Primitive datatypes

3.2.3.4 Derived datatypes

The following built-in datatypes are derived from decimal:

- integer

3.2.4 float

[Definition:] float corresponds to the IEEE single-precision 32-bit floating point type [IEEE 754–1985].

The basic value space of float consists of the values $m \times 2^e$, where $m$ is an integer whose absolute value is less than $2^{24}$, and $e$ is an integer between −149 and 104, inclusive. In addition to the basic value space described above, the value space of float also contains the following special values: positive and negative zero, positive and negative infinity and not-a-number. The order-relation on float is: $x < y$ iff $y - x$ is positive. Positive zero is greater than negative zero. Not-a-number equals itself and is greater than all float values including positive infinity.

A literal in the lexical space representing a decimal number $d$ maps to the normalized value in the value space of float that is closest to $d$ in the sense defined by [Clinger, WD (1990)]: if $d$ is exactly halfway between two such values then the even value is chosen.

3.2.4.1 Lexical representation

float values have a lexical representation consisting of a mantissa followed, optionally, by the character "E" or "e", followed by an exponent. The exponent must be an integer. The mantissa must be a decimal number. The representations for exponent and mantissa must follow the lexical rules for integer and decimal. If the "E" or "e" and the following exponent are omitted, an exponent value of 0 is assumed.

The special values positive and negative zero, positive and negative infinity and not-a-number have lexical representations 0, -0, INF, -INF and NaN, respectively.

For example, -1E4, 1267.43233E12, 12.78e−2, 12 and INF are all legal literals for float.

3.2.4.2 Canonical representation

The canonical representation for float is defined by prohibiting certain options from the Lexical representation (§3.2.4.1). Specifically, the exponent must be indicated by "E". Leading zeroes and the preceding optional "+" sign are prohibited in the exponent. For the mantissa, the preceding optional "+" sign is prohibited and the decimal point is required. For the exponent, the preceding optional "+" sign is prohibited. Leading and trailing zeroes are prohibited subject to the following: number representations must be normalized such that there is a single digit to the left of the decimal point and at least a single digit to the right of the decimal point.

3.2.4.3 Constraining facets

float has the following constraining facets:

- pattern
3.2.5 double

[Definition:] The double datatype corresponds to IEEE double-precision 64-bit floating point type [IEEE 754–1985]. The basic value space of double consists of the values \( m \times 2^e \), where \( m \) is an integer whose absolute value is less than \( 2^{53} \), and \( e \) is an integer between −1075 and 970, inclusive. In addition to the basic value space described above, the value space of double also contains the following special values: positive and negative zero, positive and negative infinity and not-a-number. The order-relation on double is: \( x < y \) iff \( y - x \) is positive. Positive zero is greater than negative zero. Not-a-number equals itself and is greater than all double values including positive infinity.

A literal in the lexical space representing a decimal number \( d \) maps to the normalized value in the value space of double that is closest to \( d \); if \( d \) is exactly halfway between two such values then the even value is chosen. This is the best approximation of \( d \) ([Clinger, WD (1990)], [Gay, DM (1990)]), which is more accurate than the mapping required by [IEEE 754–1985].

3.2.5.1 Lexical representation

double values have a lexical representation consisting of a mantissa followed, optionally, by the character "E" or "e", followed by an exponent. The exponent must be an integer. The mantissa must be a decimal number. The representations for exponent and mantissa must follow the lexical rules for integer and decimal. If the "E" or "e" and the following exponent are omitted, an exponent value of 0 is assumed.

The special values positive and negative zero, positive and negative infinity and not-a-number have lexical representations 0, −0, INF, −INF and NaN, respectively.

For example, −1E4, 1267.43233E12, 12.78e−2, 12 and INF are all legal literals for double.

3.2.5.2 Canonical representation

The canonical representation for double is defined by prohibiting certain options from the lexical representation (§3.2.5.1). Specifically, the exponent must be indicated by "E". Leading zeroes and the preceding optional "+" sign are prohibited in the exponent. For the mantissa, the preceding optional "+" sign is prohibited and the decimal point is required. For the exponent, the preceding optional "+" sign is prohibited. Leading and trailing zeroes are prohibited subject to the following: number representations must be normalized such that there is a single digit to the left of the decimal point and at least a single digit to the right of the decimal point.

3.2.5.3 Constraining facets

double has the following constraining facets:

- pattern
- enumeration
- whiteSpace
3.2.6 duration

[Definition:] duration represents a duration of time. The value space of duration is a six-dimensional space where the coordinates designate the Gregorian year, month, day, hour, minute, and second components defined in § 5.5.3.2 of [ISO 8601], respectively. These components are ordered in their significance by their order of appearance i.e. as year, month, day, hour, minute, and second.

3.2.6.1 Lexical representation

The lexical representation for duration is the [ISO 8601] extended format PnYnMnDTnHnMnS, where nY represents the number of years, nM the number of months, nD the number of days, 'T' is the date/time separator, nH the number of hours, nM the number of minutes and nS the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

The values of the Year, Month, Day, Hour and Minutes components are not restricted but allow an arbitrary integer. Similarly, the value of the Seconds component allows an arbitrary decimal. Thus, the lexical representation of duration does not follow the alternative format of § 5.5.3.2.1 of [ISO 8601].

An optional preceding minus sign ('−') is allowed, to indicate a negative duration. If the sign is omitted a positive duration is indicated. See also ISO 8601 Date and Time Formats (§D).

For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, one would write: P1Y2M3DT10H30M. One could also indicate a duration of minus 120 days as: −P120D.

Reduced precision and truncated representations of this format are allowed provided they conform to the following:

- If the number of years, months, days, hours, minutes, or seconds in any expression equals zero, the number and its corresponding designator may be omitted. However, at least one number and its designator must be present.
- The seconds part may have a decimal fraction.
- The designator 'T' shall be absent if all of the time items are absent. The designator 'P' must always be present.

For example, P1347Y, P1347M and P1Y2MT2H are all allowed; P0Y1347M and P0Y1347M0D are allowed. P−1347M is not allowed although −P1347M is allowed. P1Y2MT is not allowed.

3.2.6.2 Order relation on duration

In general, the order relation on duration is a partial order since there is no determinate relationship between certain durations such as one month (P1M) and 30 days (P30D). The order of two duration values x and y is x < y iff s+x < s+y for each qualified dateTime s in the list below. These values for s cause the greatest deviations in the addition of dateTimes and durations. Addition of durations to time instants is defined in Adding durations to dateTimes (§E).

- 1696−09−01T00:00:00Z
The following table shows the strongest relationship that can be determined between example durations. The symbol <> means that the order relation is indeterminate. Note that because of leap–seconds, a seconds field can vary from 59 to 60. However, because of the way that addition is defined in Adding durations to dateTimes (§E), they are still totally ordered.

<table>
<thead>
<tr>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1Y</td>
</tr>
<tr>
<td>&gt; P364D</td>
</tr>
<tr>
<td>P1M</td>
</tr>
<tr>
<td>&gt; P27D</td>
</tr>
<tr>
<td>P5M</td>
</tr>
<tr>
<td>&gt; P149D</td>
</tr>
</tbody>
</table>

Implementations are free to optimize the computation of the ordering relationship. For example, the following table can be used to compare durations of a small number of months against days.

| Months | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | ...
|--------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| Days   | Minimum | 28 | 59 | 89 | 120 | 150 | 181 | 212 | 242 | 273 | 303 | 334 | 365 | 393 | ...
|        | Maximum | 31 | 62 | 92 | 123 | 153 | 184 | 215 | 245 | 276 | 306 | 337 | 366 | 397 | ...

3.2.6.3 Facet Comparison for durations

In comparing duration values with minInclusive, minExclusive, maxInclusive and maxExclusive facet values indeterminate comparisons should be considered as "false".

3.2.6.4 Totally ordered durations

Certain derived datatypes of durations can be guaranteed have a total order. For this, they must have fields from only one row in the list below and the time zone must either be required or prohibited.

- year, month
- day, hour, minute, second

For example, a datatype could be defined to correspond to the [SQL] datatype Year–Month interval that required a four digit year field and a two digit month field but required all other fields to be unspecified. This datatype could be defined as below and would have a total order.

```xml
<simpleType name='SQL-Year-Month-Interval'>
  <restriction base='duration'>
    <pattern value='P\p{Nd}{4}Y\p{Nd}{2}M'/>
  </restriction>
</simpleType>
```
3.2.6.5 Constraining facets

duration has the following constraints:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.7 dateTime

[Definition:] dateTime represents a specific instant of time. The value space of dateTime is the space of combinations of date and time of day values as defined in § 5.4 of [ISO 8601].

3.2.7.1 Lexical representation

A single lexical representation, which is a subset of the lexical representations allowed by [ISO 8601], is allowed for dateTime. This lexical representation is the [ISO 8601] extended format CCYY-MM-DDThh:mm:ss where "CC" represents the century, "YY" the year, "MM" the month and "DD" the day, preceded by an optional leading "−" sign to indicate a negative number. If the sign is omitted, "+" is assumed. The letter "T" is the date/time separator and "hh", "mm", "ss" represent hour, minute and second respectively. Additional digits can be used to increase the precision of fractional seconds if desired i.e. the format ss.ss... with any number of digits after the decimal point is supported. The fractional seconds part is optional; other parts of the lexical form are not optional. To accommodate year values greater than 9999 additional digits can be added to the left of this representation. Leading zeros are required if the year value would otherwise have fewer than four digits; otherwise they are forbidden. The year 0000 is prohibited.

The CCYY field must have at least four digits, the MM, DD, SS, hh, mm and ss fields exactly two digits each (not counting fractional seconds); leading zeroes must be used if the field would otherwise have too few digits.

This representation may be immediately followed by a "Z" to indicate Coordinated Universal Time (UTC) or, to indicate the time zone, i.e. the difference between the local time and Coordinated Universal Time, immediately followed by a sign, + or −, followed by the difference from UTC represented as hh:mm (note: the minutes part is required). See ISO 8601 Date and Time Formats (§D) for details about legal values in the various fields. If the time zone is included, both hours and minutes must be present.

For example, to indicate 1:20 pm on May the 31st, 1999 for Eastern Standard Time which is 5 hours behind Coordinated Universal Time (UTC), one would write: 1999−05−31T13:20:00−05:00.

3.2.7.2 Canonical representation

The canonical representation for dateTime is defined by prohibiting certain options from the lexical representation (§3.2.7.1). Specifically, either the time zone must be omitted or, if present, the time zone must be Coordinated Universal Time (UTC) indicated by a "Z".

3.2 Primitive datatypes
3.2.7.3 Order relation on dateTime

In general, the order relation on dateTime is a partial order since there is no determinate relationship between certain instants. For example, there is no determinate ordering between (a) 2000–01–20T12:00:00 and (b) 2000–01–20T12:00:00Z. Based on timezones currently in use, (c) could vary from 2000–01–20T12:00:00+12:00 to 2000–01–20T12:00:00−13:00. It is, however, possible for this range to expand or contract in the future, based on local laws. Because of this, the following definition uses a somewhat broader range of indeterminate values: +14:00..−14:00.

The following definition uses the notation S[year] to represent the year field of S, S[month] to represent the month field, and so on. The notation (Q & “−14:00”) means adding the timezone −14:00 to Q, where Q did not already have a timezone. This is a logical explanation of the process. Actual implementations are free to optimize as long as they produce the same results.

The ordering between two dateTime P and Q is defined by the following algorithm:

A. Normalize P and Q. That is, if there is a timezone present, but it is not Z, convert it to Z using the addition operation defined in Adding durations to dateTime ($E$)

- Thus 2000–03–04T23:00:00+03:00 normalizes to 2000–03–04T20:00:00Z

B. If P and Q either both have a time zone or both do not have a time zone, compare P and Q field by field from the year field down to the second field, and return a result as soon as it can be determined. That is:

1. For each i in {year, month, day, hour, minute, second}
   1. If P[i] and Q[i] are both not specified, continue to the next i
   2. If P[i] is not specified and Q[i] is, or vice versa, stop and return P <> Q
   3. If P[i] < Q[i], stop and return P < Q
   4. If P[i] > Q[i], stop and return P > Q
2. Stop and return P = Q

C. Otherwise, if P contains a time zone and Q does not, compare as follows:

1. P < Q if P < (Q with time zone +14:00)
2. P > Q if P > (Q with time zone −14:00)
3. P <> Q otherwise, that is, if (Q with time zone +14:00) < P < (Q with time zone −14:00)

D. Otherwise, if P does not contain a time zone and Q does, compare as follows:

1. P < Q if (P with time zone −14:00) < Q.
2. P > Q if (P with time zone +14:00) > Q.
3. P <> Q otherwise, that is, if (P with time zone +14:00) < Q < (P with time zone −14:00)

Examples:

<table>
<thead>
<tr>
<th>Determinate</th>
<th>Indeterminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000–01–15T00:00:00 &lt; 2000–02–15T00:00:00</td>
<td>2000–01–01T12:00:00 &lt;&gt; 1999–12–31T23:00:00Z</td>
</tr>
</tbody>
</table>

3.2 Primitive datatypes
3.2.7.4 Totally ordered dateTimes

Certain derived types from dateTime can be guaranteed have a total order. To do so, they must require that a specific set of fields are always specified, and that remaining fields (if any) are always unspecified. For example, the date datatype without time zone is defined to contain exactly year, month, and day. Thus dates without time zone have a total order among themselves.

3.2.7.5 Constraining facets

dateTime has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.8 time

[Definition:] time represents an instant of time that recurs every day. The value space of time is the space of time of day values as defined in § 5.3 of [ISO 8601]. Specifically, it is a set of zero-duration daily time instances.

Since the lexical representation allows an optional time zone indicator, time values are partially ordered because it may not be able to determine the order of two values one of which has a time zone and the other does not. The order relation on time values is the Order relation on dateTime (§3.2.7.3) using an arbitrary date. See also Adding durations to dateTimes (§E). Pairs of time values with or without time zone indicators are totally ordered.

3.2.8.1 Lexical representation

The lexical representation for time is the left truncated lexical representation for dateTime: hh:mm:ss.sss with optional following time zone indicator. For example, to indicate 1:20 pm for Eastern Standard Time which is 5 hours behind Coordinated Universal Time (UTC), one would write: 13:20:00−05:00. See also ISO 8601 Date and Time Formats (§D).

3.2.8.2 Canonical representation

The canonical representation for time is defined by prohibiting certain options from the Lexical representation (§3.2.8.1). Specifically, either the time zone must be omitted or, if present, the time zone must be Coordinated Universal Time (UTC) indicated by a "Z". Additionally, the canonical representation for midnight is 00:00:00.

3.2 Primitive datatypes
3.2.8.3 Constraining facets

*time* has the following *constraining facets*:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.9 date

[Definition:] *date* represents a calendar date. The *value space* of *date* is the set of Gregorian calendar dates as defined in § 5.2.1 of [ISO 8601]. Specifically, it is a set of one−day long, non−periodic instances e.g. lexical 1999−10−26 to represent the calendar date 1999−10−26, independent of how many hours this day has.

Since the lexical representation allows an optional time zone indicator, *date* values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If *date* values are considered as periods of time, the order relation on *date* values is the order relation on their starting instants. This is discussed in Order relation on dateTime (§3.2.7.3). See also Adding durations to dateTimes (§E). Pairs of *date* values with or without time zone indicators are totally ordered.

3.2.9.1 Lexical representation

The lexical representation for *date* is the reduced (right truncated) lexical representation for dateTime: CCYY−MM−DD. No left truncation is allowed. An optional following time zone qualifier is allowed as for dateTime. To accommodate year values outside the range from 0001 to 9999, additional digits can be added to the left of this representation and a preceding "−" sign is allowed.

For example, to indicate May the 31st, 1999, one would write: 1999−05−31. See also ISO 8601 Date and Time Formats (§D).

3.2.9.2 Constraining facets

*date* has the following *constraining facets*:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.10 gYearMonth

[Definition:] *gYearMonth* represents a specific gregorian month in a specific gregorian year. The *value space* of *gYearMonth* is the set of Gregorian calendar months as defined in § 5.2.1 of [ISO 8601].
Specifically, it is a set of one–month long, non–periodic instances e.g. 1999–10 to represent the whole month of 1999–10, independent of how many days this month has.

Since the lexical representation allows an optional time zone indicator, gYearMonth values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If gYearMonth values are considered as periods of time, the order relation on gYearMonth values is the order relation on their starting instants. This is discussed in Order relation on dateTime (§3.2.7.3). See also Adding durations to dateTimes (§E). Pairs of gYearMonth values with or without time zone indicators are totally ordered.

**NOTE:** Because month/year combinations in one calendar only rarely correspond to month/year combinations in other calendars, values of this type are not, in general, convertible to simple values corresponding to month/year combinations in other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

### 3.2.10.1 Lexical representation

The lexical representation for gYearMonth is the reduced (right truncated) lexical representation for dateTime: CCYY–MM. No left truncation is allowed. An optional following time zone qualifier is allowed. To accommodate year values outside the range from 0001 to 9999, additional digits can be added to the left of this representation and a preceding "-" sign is allowed.

For example, to indicate the month of May 1999, one would write: 1999–05. See also ISO 8601 Date and Time Formats (§D).

### 3.2.10.2 Constraining facets

gYearMonth has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

### 3.2.11 gYear

[Definition:] gYear represents a gregorian calendar year. The value space of gYear is the set of Gregorian calendar years as defined in § 5.2.1 of [ISO 8601]. Specifically, it is a set of one–year long, non–periodic instances e.g. lexical 1999 to represent the whole year 1999, independent of how many months and days this year has.

Since the lexical representation allows an optional time zone indicator, gYear values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If gYear values are considered as periods of time, the order relation on gYear values is the order relation on their starting instants. This is discussed in Order relation on dateTime (§3.2.7.3). See also Adding durations to dateTimes (§E). Pairs of gYear values with or without time zone indicators are totally ordered.
NOTE: Because years in one calendar only rarely correspond to years in other calendars, values of this type are not, in general, convertible to simple values corresponding to years in other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

3.2.11.1 Lexical representation

The lexical representation for gYear is the reduced (right truncated) lexical representation for dateTime: CCYY. No left truncation is allowed. An optional following time zone qualifier is allowed as for dateTime. To accommodate year values outside the range from 0001 to 9999, additional digits can be added to the left of this representation and a preceding "−" sign is allowed.

For example, to indicate 1999, one would write: 1999. See also ISO 8601 Date and Time Formats (§D).

3.2.11.2 Constraining facets

gYear has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.12 gMonthDay

[Definition:] gMonthDay is a gregorian date that recurs, specifically a day of the year such as the third of May. Arbitrary recurring dates are not supported by this datatype. The value space of gMonthDay is the set of calendar dates, as defined in § 3 of [ISO 8601]. Specifically, it is a set of one−day long, annually periodic instances.

Since the lexical representation allows an optional time zone indicator, gMonthDay values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If gMonthDay values are considered as periods of time, the order relation on gMonthDay values is the order relation on their starting instants. This is discussed in Order relation on dateTime (§3.2.7.3). See also Adding durations to dateTimes (§E). Pairs of gMonthDay values with or without time zone indicators are totally ordered.

NOTE: Because day/month combinations in one calendar only rarely correspond to day/month combinations in other calendars, values of this type do not, in general, have any straightforward or intuitive representation in terms of most other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

3.2.12.1 Lexical representation

The lexical representation for gMonthDay is the left truncated lexical representation for date: −−MM−DD. An optional following time zone qualifier is allowed as for date. No preceding sign is allowed. No other formats are allowed. See also ISO 8601 Date and Time Formats (§D).
This datatype can be used to represent a specific day in a month. To say, for example, that my birthday occurs on the 14th of September ever year.

### 3.2.12.2 Constraining facets

**gMonthDay** has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- minExclusive
- maxExclusive
- minInclusive

### 3.2.13 gDay

[Definition:] **gDay** is a gregorian day that recurs, specifically a day of the month such as the 5th of the month. Arbitrary recurring days are not supported by this datatype. The value space of **gDay** is the space of a set of calendar dates as defined in § 3 of [ISO 8601]. Specifically, it is a set of one−day long, monthly periodic instances.

This datatype can be used to represent a specific day of the month. To say, for example, that I get my paycheck on the 15th of each month.

Since the lexical representation allows an optional time zone indicator, **gDay** values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If **gDay** values are considered as periods of time, the order relation on **gDay** values is the order relation on their starting instants. This is discussed in Order relation on dateTimes (§3.2.7.3). See also Adding durations to dateTimes (§E). Pairs of **gDay** values with or without time zone indicators are totally ordered.

**NOTE:** Because days in one calendar only rarely correspond to days in other calendars, values of this type do not, in general, have any straightforward or intuitive representation in terms of most other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

### 3.2.13.1 Lexical representation

The lexical representation for **gDay** is the left truncated lexical representation for **date**: −−−DD . An optional following time zone qualifier is allowed as for **date**. No preceding sign is allowed. No other formats are allowed. See also ISO 8601 Date and Time Formats (§D).

### 3.2.13.2 Constraining facets

**gDay** has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
3.2.14 gMonth

[Definition:]  **gMonth** is a gregorian month that recurs every year. The *value space* of **gMonth** is the space of a set of *calendar months* as defined in § 3 of [ISO 8601]. Specifically, it is a set of one−month long, yearly periodic instances.

This datatype can be used to represent a specific month. To say, for example, that Thanksgiving falls in the month of November.

Since the lexical representation allows an optional time zone indicator, **gMonth** values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If **gMonth** values are considered as periods of time, the order relation on **gMonth** is the order relation on their starting instants. This is discussed in Order relation on date:Time (§3.2.7.3). See also Adding durations to date:Times (§E). Pairs of **gMonth** values with or without time zone indicators are totally ordered.

   **NOTE:** Because months in one calendar only rarely correspond to months in other calendars, values of this type do not, in general, have any straightforward or intuitive representation in terms of most other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

3.2.14.1 Lexical representation

The lexical representation for **gMonth** is the left and right truncated lexical representation for date: −−MM−−. An optional following time zone qualifier is allowed as for date. No preceding sign is allowed. No other formats are allowed. See also ISO 8601 Date and Time Formats (§D).

3.2.14.2 Constraining facets

**gMonth** has the following _constraining facets:_

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.15 hexBinary

[Definition:]  **hexBinary** represents arbitrary hex−encoded binary data. The *value space* of **hexBinary** is the set of finite−length sequences of binary octets.

3.2 Primitive datatypes
3.2.15.1 Lexical Representation

**hexBinary** has a lexical representation where each binary octet is encoded as a character tuple, consisting of two hexadecimal digits ([0–9a–fA–F]) representing the octet code. For example, "0FB7" is a hex encoding for the 16–bit integer 4023 (whose binary representation is 11110110111).

3.2.15.2 Canonical Representation

The canonical representation for **hexBinary** is defined by prohibiting certain options from the Lexical Representation (§3.2.15.1). Specifically, the lower case hexadecimal digits ([a–f]) are not allowed.

3.2.15.3 Constraining facets

**hexBinary** has the following constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

3.2.16 base64Binary

[Definition:] **base64Binary** represents Base64–encoded arbitrary binary data. The value space of **base64Binary** is the set of finite–length sequences of binary octets. For **base64Binary** data the entire binary stream is encoded using the Base64 Content–Transfer–Encoding defined in Section 6.8 of [RFC 2045].

3.2.16.1 Constraining facets

**base64Binary** has the following constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

3.2.17 anyURI

[Definition:] **anyURI** represents a Uniform Resource Identifier Reference (URI). An **anyURI** value can be absolute or relative, and may have an optional fragment identifier (i.e., it may be a URI Reference). This type should be used to specify the intention that the value fulfills the role of a URI as defined by [RFC 2396], as amended by [RFC 2732].

The mapping from **anyURI** values to URIs is as defined in Section 5.4 Locator Attribute of [XML Linking Language] (see also Section 8 Character Encoding in URI References of [Character Model]). This means that a wide range of internationalized resource identifiers can be specified when an **anyURI** is called for, and still be understood as URIs per [RFC 2396], as amended by [RFC 2732], where appropriate to identify resources.
NOTE: Each URI scheme imposes specialized syntax rules for URIs in that scheme, including restrictions on the syntax of allowed fragment identifiers. Because it is impractical for processors to check that a value is a context-appropriate URI reference, this specification follows the lead of [RFC 2396] (as amended by [RFC 2732]) in this matter: such rules and restrictions are not part of type validity and are not checked by minimally conforming processors. Thus in practice the above definition imposes only very modest obligations on minimally conforming processors.

3.2.17.1 Lexical representation

The lexical space of anyURI is finite-length character sequences which, when the algorithm defined in Section 5.4 of [XML Linking Language] is applied to them, result in strings which are legal URIs according to [RFC 2396], as amended by [RFC 2732].

NOTE: Spaces are, in principle, allowed in the lexical space of anyURI, however, their use is highly discouraged (unless they are encoded by %20).

3.2.17.2 Constraining facets

anyURI has the following constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

3.2.18 QName

[Definition:] QName represents XML qualified names. The value space of QName is the set of tuples \(\{\text{namespace name}, \text{local part}\}\), where namespace name is an anyURI and local part is an NCName. The lexical space of QName is the set of strings that match the QName production of [Namespaces in XML].

NOTE: The mapping between literals in the lexical space and values in the value space of QName requires a namespace declaration to be in scope for the context in which QName is used.

3.2.18.1 Constraining facets

QName has the following constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace
3.2.19 NOTATION

[Definition:] NOTATION represents the NOTATION attribute type from [XML 1.0 (Second Edition)]. The value space of NOTATION is the set QNames. The lexical space of NOTATION is the set of all names of notations declared in the current schema.

Schema Component Constraint: enumeration facet value required for NOTATION
It is an error for NOTATION to be used directly in a schema. Only datatypes that are derived from NOTATION by specifying a value for enumeration can be used in a schema.

For compatibility (see Terminology (§1.4)) NOTATION should be used only on attributes.

3.2.19.1 Constraining facets

NOTATION has the following constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

3.3 Derived datatypes

3.3.1 normalizedString

[Definition:] normalizedString represents white space normalized strings. The value space of normalizedString is the set of strings that do not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters. The lexical space of normalizedString is the set of strings that do not contain the carriage return (#xD) nor tab (#x9) characters. The base type of normalizedString is string.

3.3.1.1 Constraining facets

normalizedString has the following constraining facets:

- length
- minLength
- maxLength
3.3.1.2 Derived datatypes

The following built-in datatypes are derived from normalizedString:

- token

3.3.2 token

[Definition:] token represents tokenized strings. The value space of token is the set of strings that do not contain the line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces. The lexical space of token is the set of strings that do not contain the line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces. The base type of token is normalizedString.

3.3.2.1 Constraining facets

token has the following constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

3.3.2.2 Derived datatypes

The following built-in datatypes are derived from token:

- language
- NMTOKEN
- Name

3.3.3 language

[Definition:] language represents natural language identifiers as defined by [RFC 1766]. The value space of language is the set of all strings that are valid language identifiers as defined in the language identification section of [XML 1.0 (Second Edition)]. The lexical space of language is the set of all strings that are valid language identifiers as defined in the language identification section of [XML 1.0 (Second Edition)]. The base type of language is token.

3.3.3.1 Constraining facets

language has the following constraining facets:

- length
- minLength
• maxLength
• pattern
• enumeration
• whiteSpace

3.3.4 NMTOKEN

[Definition:] NMTOKEN represents the NMTOKEN attribute type from [XML 1.0 (Second Edition)]. The _value space_ of NMTOKEN is the set of tokens that match the Nmtoken production in [XML 1.0 (Second Edition)]. The _lexical space_ of NMTOKEN is the set of strings that match the Nmtoken production in [XML 1.0 (Second Edition)]. The _base type_ of NMTOKEN is token.

For compatibility (see Terminology (§1.4)) NMTOKEN should be used only on attributes.

3.3.4.1 Constraining facets

NMTOKEN has the following _constraining facets_:

• length
• minLength
• maxLength
• pattern
• enumeration
• whiteSpace

3.3.4.2 Derived datatypes

The following _built-in_ datatypes are _derived_ from NMTOKEN:

• NMTOKENS

3.3.5 NMTOKENS

[Definition:] NMTOKENS represents the NMTOKENS attribute type from [XML 1.0 (Second Edition)]. The _value space_ of NMTOKENS is the set of finite, non-zero-length sequences of NMTOKEN-s. The _lexical space_ of NMTOKENS is the set of white space separated lists of tokens, of which each token is in the _lexical space_ of NMTOKEN. The _itemType_ of NMTOKENS is NMTOKEN.

For compatibility (see Terminology (§1.4)) NMTOKENS should be used only on attributes.

3.3.5.1 Constraining facets

NMTOKENS has the following _constraining facets_:

• length
• minLength
• maxLength
• enumeration
• whiteSpace
3.3.6 Name

[Definition:] **Name** represents XML Names. The *value space* of **Name** is the set of all strings which *match* the Name production of [XML 1.0 (Second Edition)]. The *lexical space* of **Name** is the set of all strings which *match* the Name production of [XML 1.0 (Second Edition)]. The *base type* of **Name** is `token`.

3.3.6.1 Constraining facets

**Name** has the following *constraining facets*:

- `length`
- `minLength`
- `maxLength`
- `pattern`
- `enumeration`
- `whiteSpace`

3.3.6.2 Derived datatypes

The following *built-in* datatypes are *derived* from **Name**:

- `NCName`

3.3.7 NCName

[Definition:] **NCName** represents XML "non-colonized" Names. The *value space* of **NCName** is the set of all strings which *match* the NCName production of [Namespaces in XML]. The *lexical space* of **NCName** is the set of all strings which *match* the NCName production of [Namespaces in XML]. The *base type* of **NCName** is **Name**.

3.3.7.1 Constraining facets

**NCName** has the following *constraining facets*:

- `length`
- `minLength`
- `maxLength`
- `pattern`
- `enumeration`
- `whiteSpace`

3.3.7.2 Derived datatypes

The following *built-in* datatypes are *derived* from **NCName**:

- `ID`
- `IDREF`
- `ENTITY`
3.3.8 ID

[Definition:]  ID represents the ID attribute type from [XML 1.0 (Second Edition)]. The -value space- of ID is the set of all strings that -match- the NCName production in [Namespaces in XML]. The -lexical space- of ID is the set of all strings that -match- the NCName production in [Namespaces in XML]. The -base type- of ID is NCName.

For compatibility (see Terminology (§1.4)) ID should be used only on attributes.

3.3.8.1 Constraining facets

ID has the following -constraining facets-:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

3.3.9 IDREF

[Definition:]  IDREF represents the IDREF attribute type from [XML 1.0 (Second Edition)]. The -value space- of IDREF is the set of all strings that -match- the NCName production in [Namespaces in XML]. The -lexical space- of IDREF is the set of strings that -match- the NCName production in [Namespaces in XML]. The -base type- of IDREF is NCName.

For compatibility (see Terminology (§1.4)) this datatype should be used only on attributes.

3.3.9.1 Constraining facets

IDREF has the following -constraining facets-:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

3.3.9.2 Derived datatypes

The following -built-in- datatypes are -derived- from IDREF:

- IDREFS

3.3.10 IDREFS

[Definition:]  IDREFS represents the IDREFS attribute type from [XML 1.0 (Second Edition)]. The -value space- of IDREFS is the set of finite, non-zero-length sequences of IDREFs. The -lexical space- of IDREFS is the set of white space separated lists of tokens, of which each token is in the -lexical space- of IDREF. The
-itemType- of IDREFS is IDREF.

For compatibility (see Terminology (§1.4)) IDREFS should be used only on attributes.

3.3.10.1 Constraining facets

IDREFS has the following -constraining facets:

- length
- minLength
- maxLength
- enumeration
- whiteSpace

3.3.11 ENTITY

[Definition:] ENTITY represents the ENTITY attribute type from [XML 1.0 (Second Edition)]. The -value space- of ENTITY is the set of all strings that -match- the NCName production in [Namespaces in XML] and have been declared as an unparsed entity in a document type definition. The -lexical space- of ENTITY is the set of all strings that -match- the NCName production in [Namespaces in XML]. The -base type- of ENTITY is NCName.

NOTE: The -value space- of ENTITY is scoped to a specific instance document.

For compatibility (see Terminology (§1.4)) ENTITY should be used only on attributes.

3.3.11.1 Constraining facets

ENTITY has the following -constraining facets:

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

3.3.11.2 Derived datatypes

The following -built-in- datatypes are -derived- from ENTITY:

- ENTITIES

3.3.12 ENTITIES

[Definition:] ENTITIES represents the ENTITIES attribute type from [XML 1.0 (Second Edition)]. The -value space- of ENTITIES is the set of finite, non-zero-length sequences of ENTITY-s that have been declared as unparsed entities in a document type definition. The -lexical space- of ENTITIES is the set of white space separated lists of tokens, of which each token is in the -lexical space- of ENTITY. The -itemType- of ENTITIES is ENTITY.
NOTE: The value space of ENTITIES is scoped to a specific instance document.

For compatibility (see Terminology (§1.4)) ENTITIES should be used only on attributes.

3.3.12.1 Constraining facets

ENTITIES has the following constraining facets:

- length
- minLength
- maxLength
- enumeration
- whiteSpace

3.3.13 integer

[Definition:] integer is derived from decimal by fixing the value of fractionDigits to be 0. This results in the standard mathematical concept of the integer numbers. The value space of integer is the infinite set \{...,−2,−1,0,1,2,...\}. The base type of integer is decimal.

3.3.13.1 Lexical representation

integer has a lexical representation consisting of a finite−length sequence of decimal digits (#x30−#x39) with an optional leading sign. If the sign is omitted, "+" is assumed. For example: −1, 0, 1267967543233, +100000.

3.3.13.2 Canonical representation

The canonical representation for integer is defined by prohibiting certain options from the Lexical representation (§3.3.13.1). Specifically, the preceding optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.13.3 Constraining facets

integer has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.3.13.4 Derived datatypes

The following built−in datatypes are derived from integer:

- nonPositiveInteger
- long

3.3 Derived datatypes
3.3.14 nonPositiveInteger

[Definition:] nonPositiveInteger is derived from integer by setting the value of maxInclusive to be 0. This results in the standard mathematical concept of the non–positive integers. The value space of nonPositiveInteger is the infinite set {..., −2, −1, 0}. The base type of nonPositiveInteger is integer.

3.3.14.1 Lexical representation

nonPositiveInteger has a lexical representation consisting of a negative sign ("−") followed by a finite–length sequence of decimal digits (#x30–#x39). If the sequence of digits consists of all zeros then the sign is optional. For example: −1, 0, −12678967543233, −100000.

3.3.14.2 Canonical representation

The canonical representation for nonPositiveInteger is defined by prohibiting certain options from the Lexical representation (§3.3.14.1). Specifically, the negative sign ("−") is required with the token "0" and leading zeroes are prohibited.

3.3.14.3 Constraining facets

nonPositiveInteger has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.3.14.4 Derived datatypes

The following built–in datatypes are derived from nonPositiveInteger:

- negativeInteger

3.3.15 negativeInteger

[Definition:] negativeInteger is derived from nonPositiveInteger by setting the value of maxInclusive to be −1. This results in the standard mathematical concept of the negative integers. The value space of negativeInteger is the infinite set {..., −2, −1}. The base type of negativeInteger is nonPositiveInteger.

3.3.15.1 Lexical representation

negativeInteger has a lexical representation consisting of a negative sign ("−") followed by a finite–length sequence of decimal digits (#x30–#x39). For example: −1, −12678967543233, −100000.
3.3.15.2 Canonical representation

The canonical representation for `negativeInteger` is defined by prohibiting certain options from the Lexical representation (§3.3.15.1). Specifically, leading zeroes are prohibited.

3.3.15.3 Constraining facets

`negativeInteger` has the following -constraining facets:

- `totalDigits`
- `fractionDigits`
- `pattern`
- `whiteSpace`
- `enumeration`
- `maxInclusive`
- `maxExclusive`
- `minInclusive`
- `minExclusive`

3.3.16 long

[Definition:] `long` is derived from `integer` by setting the value of `maxInclusive` to be 9223372036854775807 and `minInclusive` to be −9223372036854775808. The base type of `long` is `integer`.

3.3.16.1 Lexical representation

`long` has a lexical representation consisting of an optional sign followed by a finite–length sequence of decimal digits (#x30–#x39). If the sign is omitted, "+" is assumed. For example: −1, 0, 12678967543233, +100000.

3.3.16.2 Canonical representation

The canonical representation for `long` is defined by prohibiting certain options from the Lexical representation (§3.3.16.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.16.3 Constraining facets

`long` has the following -constraining facets:

- `totalDigits`
- `fractionDigits`
- `pattern`
- `whiteSpace`
- `enumeration`
- `maxInclusive`
- `maxExclusive`
- `minInclusive`
- `minExclusive`
3.3.16.4 Derived datatypes

The following _built-in_ datatypes are _derived_ from _long_:

- `int`

3.3.17 `int`

[Definition:] `int` is _derived_ from _long_ by setting the value of _maxInclusive_ to be 2147483647 and _minInclusive_ to be −2147483648. The _base type_ of `int` is _long_.

3.3.17.1 Lexical representation

`int` has a lexical representation consisting of an optional sign followed by a finite–length sequence of decimal digits (#x30–#x39). If the sign is omitted, "+" is assumed. For example: −1, 0, 126789675, +100000.

3.3.17.2 Canonical representation

The canonical representation for `int` is defined by prohibiting certain options from the _Lexical representation_ (§3.3.17.1). Specifically, the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.17.3 Constraining facets

`int` has the following _constraining facets_:

- `totalDigits`
- `fractionDigits`
- `pattern`
- `whiteSpace`
- `enumeration`
- `maxInclusive`
- `maxExclusive`
- `minInclusive`
- `minExclusive`

3.3.17.4 Derived datatypes

The following _built-in_ datatypes are _derived_ from `int`:

- `short`

3.3.18 `short`

[Definition:] `short` is _derived_ from `int` by setting the value of _maxInclusive_ to be 32767 and _minInclusive_ to be −32768. The _base type_ of `short` is `int`.

3.3.18.1 Lexical representation

`short` has a lexical representation consisting of an optional sign followed by a finite–length sequence of decimal digits (#x30–#x39). If the sign is omitted, "+" is assumed. For example: −1, 0, 12678, +10000.
3.3.18.2 Canonical representation

The canonical representation for short is defined by prohibiting certain options from the Lexical representation (§3.3.18.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.18.3 Constraining facets

short has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.3.18.4 Derived datatypes

The following built-in datatypes are derived from short:

- byte

3.3.19 byte

[Definition:] byte is derived from short by setting the value of maxInclusive to be 127 and minInclusive to be −128. The base type of byte is short.

3.3.19.1 Lexical representation

byte has a lexical representation consisting of an optional sign followed by a finite-length sequence of decimal digits (#x30−#x39). If the sign is omitted, "+" is assumed. For example: −1, 0, 126, +100.

3.3.19.2 Canonical representation

The canonical representation for byte is defined by prohibiting certain options from the Lexical representation (§3.3.19.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.19.3 Constraining facets

byte has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
3.3.20 nonNegativeInteger

[Definition:] nonNegativeInteger is derived from integer by setting the value of minInclusive to be 0. This results in the standard mathematical concept of the non-negative integers. The value space of nonNegativeInteger is the infinite set \( \{0, 1, 2, \ldots\} \). The base type of nonNegativeInteger is integer.

3.3.20.1 Lexical representation

nonNegativeInteger has a lexical representation consisting of an optional sign followed by a finite-length sequence of decimal digits (#x30–#x39). If the sign is omitted, "+" is assumed. For example: 1, 0, 12678967543233, +100000.

3.3.20.2 Canonical representation

The canonical representation for nonNegativeInteger is defined by prohibiting certain options from the Lexical representation (§3.3.20.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.20.3 Constraining facets

nonNegativeInteger has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.3.20.4 Derived datatypes

The following built-in datatypes are derived from nonNegativeInteger:

- unsignedLong
- positiveInteger

3.3.21 unsignedLong

[Definition:] unsignedLong is derived from nonNegativeInteger by setting the value of maxInclusive to be 18446744073709551615. The base type of unsignedLong is nonNegativeInteger.

3.3.21.1 Lexical representation

unsignedLong has a lexical representation consisting of a finite-length sequence of decimal digits (#x30–#x39). For example: 0, 12678967543233, 100000.
3.3.21.2 Canonical representation

The canonical representation for unsignedLong is defined by prohibiting certain options from the Lexical representation (§3.3.21.1). Specifically, leading zeroes are prohibited.

3.3.21.3 Constraining facets

unsignedLong has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.3.21.4 Derived datatypes

The following built-in datatypes are derived from unsignedLong:

- unsignedInt

3.3.22 unsignedInt

[Definition:] unsignedInt is derived from unsignedLong by setting the value of maxInclusive to be 4294967295. The base type of unsignedInt is unsignedLong.

3.3.22.1 Lexical representation

unsignedInt has a lexical representation consisting of a finite-length sequence of decimal digits (#x30–#x39). For example: 0, 1267896754, 100000.

3.3.22.2 Canonical representation

The canonical representation for unsignedInt is defined by prohibiting certain options from the Lexical representation (§3.3.22.1). Specifically, leading zeroes are prohibited.

3.3.22.3 Constraining facets

unsignedInt has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
• minExclusive

3.3.22.4 Derived datatypes

The following built-in datatypes are derived from unsignedInt:

• unsignedShort

3.3.23 unsignedShort

[Definition:] unsignedShort is derived from unsignedInt by setting the value of maxInclusive to be 65535. The base type of unsignedShort is unsignedInt.

3.3.23.1 Lexical representation

unsignedShort has a lexical representation consisting of a finite–length sequence of decimal digits (#x30–#x39). For example: 0, 12678, 10000.

3.3.23.2 Canonical representation

The canonical representation for unsignedShort is defined by prohibiting certain options from the Lexical representation (§3.3.23.1). Specifically, the leading zeroes are prohibited.

3.3.23.3 Constraining facets

unsignedShort has the following constraining facets:

• totalDigits
• fractionDigits
• pattern
• whiteSpace
• enumeration
• maxInclusive
• maxExclusive
• minInclusive
• minExclusive

3.3.23.4 Derived datatypes

The following built-in datatypes are derived from unsignedShort:

• unsignedByte

3.3.24 unsignedByte

[Definition:] unsignedByte is derived from unsignedShort by setting the value of maxInclusive to be 255. The base type of unsignedByte is unsignedShort.

3.3.24.1 Lexical representation

unsignedByte has a lexical representation consisting of a finite–length sequence of decimal digits (#x30–#x39). For example: 0, 126, 100.

3.3 Derived datatypes
3.3.24.2 Canonical representation

The canonical representation for unsignedByte is defined by prohibiting certain options from the Lexical representation (§3.3.24.1). Specifically, leading zeroes are prohibited.

3.3.24.3 Constraining facets

unsignedByte has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.3.25 positiveInteger

[Definition:] positiveInteger is derived from nonNegativeInteger by setting the value of minInclusive to be 1. This results in the standard mathematical concept of the positive integer numbers. The value space of positiveInteger is the infinite set {1,2,...}. The base type of positiveInteger is nonNegativeInteger.

3.3.25.1 Lexical representation

positiveInteger has a lexical representation consisting of an optional positive sign ("+") followed by a finite-length sequence of decimal digits (#x30–#x39). For example: 1, 12678967543233, +100000.

3.3.25.2 Canonical representation

The canonical representation for positiveInteger is defined by prohibiting certain options from the Lexical representation (§3.3.25.1). Specifically, the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.25.3 Constraining facets

positiveInteger has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
4 Datatype components

The following sections provide full details on the properties and significance of each kind of schema component involved in datatype definitions. For each property, the kinds of values it is allowed to have is specified. Any property not identified as optional is required to be present; optional properties which are not present have absent as their value. Any property identified as a having a set, subset or list value may have an empty value unless this is explicitly ruled out: this is not the same as absent. Any property value identified as a superset or a subset of some set may be equal to that set, unless a proper superset or subset is explicitly called for.

For more information on the notion of datatype (schema) components, see Schema Component Details of [XML Schema Part 1: Structures].

4.1 Simple Type Definition

4.1.1 The Simple Type Definition Schema Component

Simple Type definitions provide for:

- Establishing the value space and lexical space of a datatype, through the combined set of constraining facet-s specified in the definition;
- Attaching a unique name (actually a QName) to the value space and lexical space.

4.1.1 The Simple Type Definition Schema Component

The Simple Type Definition schema component has the following properties:

Schema Component: Simple Type Definition

{name}
Optional. An NCName as defined by [Namespaces in XML].

{target namespace}
Either absent or a namespace name, as defined in [Namespaces in XML].

{variety}
One of {atomic, list, union}. Depending on the value of {variety}, further properties are defined as follows:

atomic
{primitive type definition}
A built-in, primitive datatype definition (or the simple ur-type definition).

list
{item type definition}
An atomic- or union- simple type definition.

union
{member type definitions}
A non-empty sequence of simple type definitions.

{facets}
A possibly empty set of Facets (§2.4).
Datatypes are identified by their {name} and {target namespace}. Except for anonymous datatypes (those with no {name}), datatype definitions must be uniquely identified within a schema.

If {variety} is ·atomic· then the ·value space· of the datatype defined will be a subset of the ·value space· of {base type definition} (which is a subset of the ·value space· of {primitive type definition}). If {variety} is ·list· then the ·value space· of the datatype defined will be the set of finite-length sequence of values from the ·value space· of {item type definition}. If {variety} is ·union· then the ·value space· of the datatype defined will be the union of the ·value space·s of each datatype in {member type definitions}.

If {variety} is ·atomic· then the {variety} of {base type definition} must be ·atomic·. If {variety} is ·list· then the {variety} of {item type definition} must be either ·atomic· or ·union·. If {variety} is ·union· then {member type definitions} must be a list of datatype definitions.

The value of {facets} consists of the set of ·facet·s specified directly in the datatype definition unioned with the possibly empty set of {facets} of {base type definition}.

The value of {fundamental facets} consists of the set of ·fundamental facet·s and their values.

If {final} is the empty set then the type can be used in deriving other types; the explicit values restriction, list and union prevent further derivations by ·restriction·, ·list· and ·union· respectively.

**4.1.2 XML Representation of Simple Type Definition Schema Components**

The XML representation for a Simple Type Definition schema component is a `<simpleType>` element information item. The correspondences between the properties of the information item and properties of the component are as follows:

**XML Representation Summary: simpleType Element Information Item**

```
<simpleType
  final = (#all | (list | union | restriction))
  id = ID
  name = NCName
  (any attributes with non–schema namespace . . .)>
  Content:  (annotation?, (restriction | list | union))
</simpleType>
```

**Datatype Definition Schema Component**
Property Representation

{name} The actual value of the name [attribute], if present, otherwise null

{final} A set corresponding to the actual value of the final [attribute], if present, otherwise of the actual value of the finalDefault [attribute] the ancestor schema element information item, if present, otherwise the empty string, as follows:

-the empty string
-the empty set;
-#a11 {restriction, list, union};
 otherwise
 a set with members drawn from the set above, each being present or absent depending on whether the string contains an equivalently named space-delimited substring.

NOTE: Although the finalDefault [attribute] of schema may include values other than restriction, list or union, those values are ignored in the determination of {final}

{target namespace} The actual value of the targetNamespace [attribute] of the parent schema element information item.

{annotation} The annotation corresponding to the <annotation> element information item in the [children], if present, otherwise null

A derived datatype can be derived from a primitive datatype or another derived datatype by one of three means: by restriction, by list or by union.

4.1.2.1 Derivation by restriction

XML Representation Summary: restriction Element Information Item

<restriction
 base = QName
 id = ID
 {any attributes with non-schema namespace . . .}>
 Content: (annotation?, (simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive | totalDigits | fractionDigits | length | minLength | maxLength | enumeration | whiteSpace | pattern)*))
</restriction>

Simple Type Definition Schema Component

Property Representation

{variety} The actual value of {variety} of {base type definition}

{facets} The union of the set of Facets (§2.4) components resolved to by the facet [children] merged with [facets] from {base type definition}, subject to the Facet Restriction Valid constraints specified in Facets (§2.4).

{base type} The Simple Type Definition component resolved to by the actual value of the base

4.1 Simple Type Definition
Example An electronic commerce schema might define a datatype called `Sku` (the barcode number that appears on products) from the _built-in_ datatype `string` by supplying a value for the _pattern_ facet.

```xml
<simpleType name='Sku'>
  <restriction base='string'>
    <pattern value='\d{3}−[A−Z]{2}'/>
  </restriction>
</simpleType>
```

In this case, `Sku` is the name of the new _user-derived_ datatype, `string` is its _base type_ and _pattern_ is the facet.

### 4.1.2.2 Derivation by list

XML Representation Summary: `list` Element Information Item

```xml
<list
  id = ID
  itemType = QName
  {any attributes with non-schema namespace . . .}>
  Content:  (annotation?, (simpleType?))
</list>
```

**Simple Type Definition Schema Component**

**Property**      **Representation**

*{variety}*       list

*{item type definition}* [attribute] or the `<simpleType>` [children], whichever is present.

A _list_ datatype must be _derived_ from an _atomic_ or a _union_ datatype, known as the _itemType_ of the _list_ datatype. This yields a datatype whose _value space_ is composed of finite-length sequences of values from the _value space_ of the _itemType_ and whose _lexical space_ is composed of white space separated lists of literals of the _itemType_.

Example A system might want to store lists of floating point values.

```xml
<simpleType name='listOfFloat'>
  <list itemType='float'/>  ...
</simpleType>
```

In this case, `listOfFloat` is the name of the new _user-derived_ datatype, `float` is its _itemType_ and _list_ is the derivation method.

As mentioned in _List datatypes_ (§2.5.1.2), when a datatype is _derived_ from a _list_ datatype, the following _constraining facet-s_ can be used:

- _length_
- _maxLength_
- _minLength_

4.1 Simple Type Definition
regardless of the constraining facets that are applicable to the atomic datatype that serves as the itemType of the list.

For each of length, maxLength and minLength, the unit of length is measured in number of list items. The value of whiteSpace is fixed to the value collapse.

4.1.2.3 Derivation by union

XML Representation Summary: union Element Information Item

```
<union
  id = ID
  memberTypes = List of QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType*))
</union>
```

Simple Type Definition Schema Component

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{variety}</td>
<td>union</td>
</tr>
</tbody>
</table>
| {member type definitions}     | The sequence of Simple Type Definition components resolved to by the items in the actual value of the memberTypes [attribute], if any, in order, followed by the Simple Type Definition components resolved to by the <simpleType> [children], if any, in order. If {variety} is union for any Simple Type Definition components resolved to above, then the that Simple Type Definition is replaced by its {member type definitions}.

A union datatype can be derived from one or more atomic, list or other union datatypes, known as the memberTypes of that union datatype.

Example As an example, taken from a typical display oriented text markup language, one might want to express font sizes as an integer between 8 and 72, or with one of the tokens "small", "medium" or "large". The union type definition below would accomplish that.

```
<xsd:attribute name="size">
  <xsd:simpleType>
    <xsd:union>
      <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
          <xsd:minInclusive value="8"/>
          <xsd:maxInclusive value="72"/>
        </xsd:restriction>
      </xsd:simpleType>
      <xsd:simpleType>
        <xsd:restriction base="xsd:NMTOKEN">
          <xsd:enumeration value="small"/>
          <xsd:enumeration value="medium"/>
          <xsd:enumeration value="large"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:union>
  </xsd:simpleType>
</xsd:attribute>
```
As mentioned in Union datatypes (§2.5.1.3), when a datatype is derived from a union datatype, the only following constraining facets can be used:

- pattern
- enumeration

regardless of the constraining facets that are applicable to the datatypes that participate in the union.

4.1.3 Constraints on XML Representation of Simple Type Definition

Schema Representation Constraint: Single Facet Value
Unless otherwise specifically allowed by this specification (Multiple patterns (§4.3.4.3) and Multiple enumerations (§4.3.5.3)) any given constraining facet can only be specified once within a single derivation step. Schema Representation Constraint: itemType attribute or simpleType child
Either the itemType [attribute] or the <simpleType> [child] of the <list> element must be present, but not both. Schema Representation Constraint: base attribute or simpleType child
Either the base [attribute] or the simpleType [child] of the <restriction> element must be present, but not both. Schema Representation Constraint: memberTypes attribute or simpleType children
Either the memberTypes [attribute] of the <union> element must be non-empty or there must be at least one simpleType [child].

4.1.4 Simple Type Definition Validation Rules

Validation Rule: Facet Valid
A value in a value space is facet–valid with respect to a constraining facet component if: 1 the value is facet–valid with respect to the particular constraining facet as specified below. Validation Rule: Datatype Valid
A string is datatype–valid with respect to a datatype definition if: 1 it matches a literal in the lexical space of the datatype, determined as follows: 1.1 if pattern is a member of {facets}, then the string must be pattern valid (§4.3.4.4); 1.2 if pattern is not a member of {facets}, then 1.2.1 if {variety} is atomic then the string must match a literal in the lexical space of {base type definition}; 1.2.2 if {variety} is list then the string must be a sequence of white space separated tokens, each of which matches a literal in the lexical space of {item type definition}; 1.2.3 if {variety} is union then the string must match a literal in the lexical space of at least one member of {member type definitions} the value denoted by the literal matched in the previous step is a member of the value space of the datatype, as determined by it being Facet Valid (§4.1.4) with respect to each member of {facets} (except for pattern).
### 4.1.5 Constraints on Simple Type Definition Schema Components

#### Schema Component Constraint: applicable facets

The constraining facets which are allowed to be members of `{facets}` are dependent on `{base type definition}` as specified in the following table:

<table>
<thead>
<tr>
<th>{base type definition}</th>
<th>applicable {facets}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>If <code>{variety}</code> is list, then</strong></td>
<td></td>
</tr>
<tr>
<td>[all datatypes]</td>
<td>length, minLength, maxLength, pattern, enumeration, whiteSpace</td>
</tr>
<tr>
<td><strong>If <code>{variety}</code> is union, then</strong></td>
<td></td>
</tr>
<tr>
<td>[all datatypes]</td>
<td>pattern, enumeration</td>
</tr>
<tr>
<td><strong>else if <code>{variety}</code> is atomic, then</strong></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>length, minLength, maxLength, pattern, enumeration, whiteSpace</td>
</tr>
<tr>
<td>boolean</td>
<td>pattern, whiteSpace</td>
</tr>
<tr>
<td>float</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>double</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>decimal</td>
<td>totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>dateTime</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>time</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>date</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>gYear</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>gDay</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>gMonth</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>hexBinary</td>
<td>length, minLength, maxLength, pattern, enumeration, whiteSpace</td>
</tr>
<tr>
<td>base64Binary</td>
<td>length, minLength, maxLength, pattern, enumeration, whiteSpace</td>
</tr>
<tr>
<td>anyURI</td>
<td>length, minLength, maxLength, pattern, enumeration, whiteSpace</td>
</tr>
<tr>
<td>QName</td>
<td>length, minLength, maxLength, pattern, enumeration, whiteSpace</td>
</tr>
</tbody>
</table>
### 4.1 Simple Type Definition for anySimpleType

There is a simple type definition nearly equivalent to the simple version of the ur–type definition present in every schema by definition. It has the following properties:

#### Schema Component: anySimpleType

- **{name}**: anySimpleType
- **{target namespace}**: http://www.w3.org/2001/XMLSchema
- **{basetype definition}**: the ur–type definition
- **{final}**: the empty set
- **{variety}**: absent

### 4.2 Fundamental Facets

#### 4.2.1 equal

Every -value space- supports the notion of equality, with the following rules:

- for any \(a\) and \(b\) in the -value space-, either \(a\) is equal to \(b\), denoted \(a = b\), or \(a\) is not equal to \(b\), denoted \(a \neq b\).
- there is no pair \(a\) and \(b\) from the -value space- such that both \(a = b\) and \(a \neq b\).
- for all \(a\) in the -value space-, \(a = a\).
- for any \(a\) and \(b\) in the -value space-, \(a = b\) if and only if \(b = a\).
- for any \(a, b\) and \(c\) in the -value space-, if \(a = b\) and \(b = c\), then \(a = c\).
- for any \(a\) and \(b\) in the -value space- if \(a = b\), then \(a\) and \(b\) cannot be distinguished (i.e., equality is identity).

Note that a consequence of the above is that, given -value space- \(A\) and -value space- \(B\) where \(A\) and \(B\) are not related by -restriction- or -union-, for every pair of values \(a\) from \(A\) and \(b\) from \(B\), \(a \neq b\).

On every datatype, the operation Equal is defined in terms of the equality property of the -value space-: for any values \(a, b\) drawn from the -value space-, \(Equal(a, b)\) is true if \(a = b\), and false otherwise.

**NOTE**: There is no schema component corresponding to the equal -fundamental facet-.
4.2.2 ordered

[Definition:] An order relation on a value space is a mathematical relation that imposes a total order or a partial order on the members of the value space.

[Definition:] A value space, and hence a datatype, is said to be ordered if there exists an order−relation defined for that value space.

[Definition:] A partial order is an order−relation that is irreflexive, asymmetric and transitive.

A partial order has the following properties:

- for no a in the value space, a < a (irreflexivity)
- for all a and b in the value space, a < b implies not(b < a) (asymmetry)
- for all a, b and c in the value space, a < b and b < c implies a < c (transitivity)

The notation a <> b is used to indicate the case when a != b and neither a < b nor b < a

[Definition:] A total order is an partial order such that for no a and b is it the case that a <> b.

A total order has all of the properties specified above for partial order, plus the following property:

- for all a and b in the value space, either a < b or b < a or a = b

NOTE: The fact that this specification does not define an order−relation for some datatype does not mean that some other application cannot treat that datatype as being ordered by imposing its own order relation.

ordered provides for:

- indicating whether an order−relation is defined on a value space, and if so, whether that order−relation is a partial order, or a total order.

4.2.2.1 The ordered Schema Component

Schema Component: ordered

{value} One of {false, partial, total}.

{value} depends on {variety}, {facets} and {member type definitions} in the Simple Type Definition component in which a ordered component appears as a member of {fundamental facets}.

When {variety} is atomic, {value} is inherited from {value} of {base type definition}. For all primitive types {value} is as specified in the table in Fundamental Facets (§C.1).

When {variety} is list, {value} is false.

When {variety} is union, if {value} is true for every member of {member type definitions} and all members of {member type definitions} share a common ancestor, then {value} is true; else {value} is false.
4.2.3 bounded

[Definition:] A value \( u \) in an ordered value space \( U \) is said to be an inclusive upper bound of a value space \( V \) (where \( V \) is a subset of \( U \)) if for all \( v \) in \( V \), \( u \geq v \).

[Definition:] A value \( u \) in an ordered value space \( U \) is said to be an exclusive upper bound of a value space \( V \) (where \( V \) is a subset of \( U \)) if for all \( v \) in \( V \), \( u > v \).

[Definition:] A value \( l \) in an ordered value space \( L \) is said to be an inclusive lower bound of a value space \( V \) (where \( V \) is a subset of \( L \)) if for all \( v \) in \( V \), \( l \leq v \).

[Definition:] A value \( l \) in an ordered value space \( L \) is said to be an exclusive lower bound of a value space \( V \) (where \( V \) is a subset of \( L \)) if for all \( v \) in \( V \), \( l < v \).

[Definition:] A datatype is bounded if its value space has either an inclusive upper bound, or an exclusive upper bound, and either an inclusive lower bound, and an exclusive lower bound.

bounded provides for:

- indicating whether a value space is bounded.

4.2.3.1 The bounded Schema Component

Schema Component: bounded

\n
\{value\}

A boolean.

\{value\} depends on \{variety\}, \{facets\} and \{member type definitions\} in the Simple Type Definition component in which a bounded component appears as a member of \{fundamental facets\}.

When \{variety\} is atomic, if one of \-minInclusive\, or \-minExclusive\, and one of \-maxInclusive\, or \-maxExclusive\, are among \{facets\}, then \{value\} is true; else \{value\} is false.

When \{variety\} is list, if \-length\, or both of \-minLength\, and \-maxLength\, are among \{facets\}, then \{value\} is true; else \{value\} is false.

When \{variety\} is union, if \{value\} is true for every member of \{member type definitions\} and all members of \{member type definitions\} share a common ancestor, then \{value\} is true; else \{value\} is false.

4.2.4 cardinality

[Definition:] Every value space has associated with it the concept of cardinality. Some value spaces are finite, some are countably infinite while still others could conceivably be uncountably infinite (although no value space defined by this specification is uncountably infinite). A datatype is said to have the cardinality of its value space.

It is sometimes useful to categorize value spaces (and hence, datatypes) as to their cardinality. There are two significant cases:

- value spaces that are finite
- **_value space_**s that are countably infinite

- **cardinality** provides for:
  - indicating whether the **cardinality** of a **value space** is _finite_ or _countably infinite_

### 4.2.4.1 The cardinality Schema Component

**Schema Component:** cardinality

**{value}**

One of _{finite, countably infinite}_.

{value} depends on {variety}, {facets} and {member type definitions} in the Simple Type Definition component in which a **cardinality** component appears as a member of _{fundamental facets}_.

When {variety} is _atomic_ and {value} of {base type definition} is _finite_, then {value} is _finite_.

When {variety} is _atomic_ and {value} of {base type definition} is _countably infinite_ and either of the following conditions are true, then {value} is _finite_; else {value} is _countably infinite_:

1. one of _length_, _maxLength_, _totalDigits_ is among {facets},
2. all of the following are true:
   1. one of _minInclusive_ or _minExclusive_ is among {facets}
   2. one of _maxInclusive_ or _maxExclusive_ is among {facets}
   3. either of the following are true:
      1. _fractionDigits_ is among {facets}
      2. _base type definition_ is one of _date_, _gYearMonth_, _gYear_, _gMonthDay_, _gDay_ or _gMonth_ or any type derived from them

When {variety} is _list_, if _length_ or both of _minLength_ and _maxLength_ are among {facets}, then {value} is _finite_; else {value} is _countably infinite_.

When {variety} is _union_, if {value} is _finite_ for every member of _{member type definitions}_, then {value} is _finite_; else {value} is _countably infinite_.

### 4.2.5 numeric

[Definition:] A datatype is said to be **numeric** if its values are conceptually quantities (in some mathematical number system).

[Definition:] A datatype whose values are not **numeric** is said to be **non–numeric**.

- **numeric** provides for:
  - indicating whether a **value space** is **numeric**.

#### 4.2.5.1 The numeric Schema Component

**Schema Component:** numeric

---

4.2 Fundamental Facets 268
A boolean

{value} depends on {variety}, {facets}, {base type definition} and {member type definitions} in the Simple Type Definition component in which a cardinality component appears as a member of {fundamental facets}.

When {variety} is atomic, {value} is inherited from {value} of {base type definition}. For all primitive types, {value} is as specified in the table in Fundamental Facets (§C.1).

When {variety} is list, {value} is false.

When {variety} is union, if {value} is true for every member of {member type definitions}, then {value} is true; else {value} is false.

### 4.3 Constraining Facets

4.3.1 length

[Definition:] length is the number of units of length, where units of length varies depending on the type that is being derived from. The value of length must be a nonNegativeInteger.

For string and datatypes derived from string, length is measured in units of characters as defined in [XML 1.0 (Second Edition)]. For anyURI, length is measured in units of characters (as for string). For hexBinary and base64Binary and datatypes derived from them, length is measured in octets (8 bits) of binary data. For datatypes derived by list, length is measured in number of list items.

NOTE: For string and datatypes derived from string, length will not always coincide with "string length" as perceived by some users or with the number of storage units in some digital representation. Therefore, care should be taken when specifying a value for length and in attempting to infer storage requirements from a given value for length.

-length provides for:

- Constraining a value space to values with a specific number of units of length, where units of length varies depending on {base type definition}.

Example The following is the definition of a user-derived datatype to represent product codes which must be exactly 8 characters in length. By fixing the value of the length facet we ensure that types derived from productCode can change or set the values of other facets, such as pattern, but cannot change the length.

```xml
<simpleType name='productCode'>
  <restriction base='string'>
    <length value='8' fixed='true'/>
  </restriction>
</simpleType>
```
4.3.1.1 The length Schema Component

Schema Component: length

{value}  A nonNegativeInteger.
{fixed}  A boolean.
{annotation}  Optional. An annotation.

If {fixed} is true, then types for which the current type is the {base type definition} cannot specify a value for length other than {value}.

4.3.1.2 XML Representation of length Schema Components

The XML representation for a length schema component is a <length> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: length Element Information Item

<length
  fixed = boolean : false
  id = ID__
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}>
  Content:  (annotation?)
</length>

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value &lt;attribute&gt;</td>
</tr>
<tr>
<td>{fixed}</td>
<td>The actual value of the fixed &lt;attribute&gt;, if present, otherwise false</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the &lt;children&gt;, if any.</td>
</tr>
</tbody>
</table>

4.3.1.3 length Validation Rules

Validation Rule: Length Valid

A value in a ·value space· is facet–valid with respect to ·length·, determined as follows:1 if the {variety} is ·atomic· then1.1 if {primitive type definition} is string, then the length of the value, as measured in characters ·must· be equal to {value};1.2 if {primitive type definition} is hexBinary or base64Binary, then the length of the value, as measured in octets of the binary data, ·must· be equal to {value};2 if the {variety} is ·list·, then the length of the value, as measured in list items, ·must· be equal to {value}. |
4.3.1.4 Constraints on length Schema Components

Schema Component Constraint: length and minLength or maxLength
It is an error for both length and either of minLength or maxLength to be members of {facets}. Schema Component Constraint: length valid restriction
It is an error if length is among the members of {facets} of {base type definition} and {value} is not equal to the {value} of the parent length.

4.3.2 minLength

[Definition:] minLength is the minimum number of units of length, where units of length varies depending on the type that is being derived from. The value of minLength must be a nonNegativeInteger.

For string and datatypes derived from string, minLength is measured in units of characters as defined in [XML 1.0 (Second Edition)]. For hexBinary and base64Binary and datatypes derived from them, minLength is measured in octets (8 bits) of binary data. For datatypes derived by list, minLength is measured in number of list items.

NOTE: For string and datatypes derived from string, minLength will not always coincide with "string length" as perceived by some users or with the number of storage units in some digital representation. Therefore, care should be taken when specifying a value for minLength and in attempting to infer storage requirements from a given value for minLength.

.minLength provides for:

• Constraining a value space to values with at least a specific number of units of length, where units of length varies depending on {base type definition}.

Example The following is the definition of a user-derived datatype which requires strings to have at least one character (i.e., the empty string is not in the value space of this datatype).

```xml
<simpleType name='non-empty-string'>
  <restriction base='string'>
    <minLength value='1'/>
  </restriction>
</simpleType>
```

4.3.2.1 The minLength Schema Component

Schema Component: minLength

{value}
A nonNegativeInteger.

{fixed}
A boolean.

{annotation}
Optional. An annotation.

If {fixed} is true, then types for which the current type is the {base type definition} cannot specify a value for minLength other than {value}.
4.3.2.2 XML Representation of minLength Schema Component

The XML representation for a minLength schema component is a <minLength> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: minLength Element Information Item

<minLength
   fixed = boolean : false
   id = ID
   value = nonNegativeInteger
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?)
</minLength>

**minLength Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value [attribute]</td>
</tr>
<tr>
<td>{fixed}</td>
<td>The actual value of the fixed [attribute], if present, otherwise false</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

4.3.2.3 minLength Validation Rules

**Validation Rule: minLength Valid**

A value in a ·value space· is facet–valid with respect to minLength, determined as follows: 1 if the {variety} is ·atomic·, then 1.1 if {primitive type definition} is string, then the length of the value, as measured in characters ·must· be greater than or equal to {value}; 1.2 if {primitive type definition} is hexBinary or base64Binary, then the length of the value, as measured in octets of the binary data, ·must· be greater than or equal to {value}; 2 if the {variety} is ·list·, then the length of the value, as measured in list items, ·must· be greater than or equal to {value}.

4.3.2.4 Constraints on minLength Schema Components

**Schema Component Constraint: minLength <= maxLength**

If both minLength and maxLength are members of {facets}, then the {value} of minLength ·must· be less than or equal to the {value} of maxLength. **Schema Component Constraint: minLength valid restriction**

It is an error if minLength is among the members of {facets} of {base type definition} and {value} is less than the {value} of the parent minLength.

4.3.3 maxLength

**Definition:** maxLength is the maximum number of units of length, where units of length varies depending on the type that is being derived from. The value of maxLength ·must· be a nonNegativeInteger.

For string and datatypes ·derived· from string, maxLength is measured in units of characters as defined in [XML 1.0 (Second Edition)]. For hexBinary and base64Binary and datatypes ·derived· from them, maxLength is measured in octets (8 bits) of binary data. For datatypes ·derived· by ·list·, maxLength is
NOTE: For string and datatypes derived from string, `maxLength` will not always coincide with "string length" as perceived by some users or with the number of storage units in some digital representation. Therefore, care should be taken when specifying a value for `maxLength` and in attempting to infer storage requirements from a given value for `maxLength`.

`maxLength` provides for:

- Constraining a value space to values with at most a specific number of units of length, where units of length varies depending on [base type definition].

Example The following is the definition of a user-derived datatype which might be used to accept form input with an upper limit to the number of characters that are acceptable.

```xml
<simpleType name='form-input'>
  <restriction base='string'>
    <maxLength value='50'/>
  </restriction>
</simpleType>
```

### 4.3.3.1 The maxLength Schema Component

Schema Component: `maxLength`

- `{value}`
  - A nonNegativeInteger.
- `{fixed}`
  - A boolean.
- `{annotation}`
  - Optional. An annotation.

If `{fixed}` is true, then types for which the current type is the [base type definition] cannot specify a value for `maxLength` other than `{value}`.

### 4.3.3.2 XML Representation of maxLength Schema Components

The XML representation for a maxLength schema component is a `<maxLength>` element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `maxLength` Element Information Item

```xml
<maxLength
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}/>
Content:  (annotation?)
</maxLength>
```
**maxLength Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value [attribute]</td>
</tr>
<tr>
<td>{fixed}</td>
<td>The actual value of the fixed [attribute], if present, otherwise false</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

### 4.3.3.3 maxLength Validation Rules

**Validation Rule: maxLength Valid**

A value in a *value space* is facet–valid with respect to *maxLength*, determined as follows:

1. If the {variety} is *atomic*, then:
   1.1 If {primitive type definition} is string, then the length of the value, as measured in characters, *must* be less than or equal to {value};
   1.2 If {primitive type definition} is hexBinary or base64Binary, then the length of the value, as measured in octets of the binary data, *must* be less than or equal to {value};

2. If the {variety} is *list*, then the length of the value, as measured in list items, *must* be less than or equal to {value}.

### 4.3.3.4 Constraints on maxLength Schema Components

**Schema Component Constraint: maxLength valid restriction**

It is an *error* if maxLength is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent maxLength.

### 4.3.4 pattern

**Definition:** *pattern* is a constraint on the *value space* of a datatype which is achieved by constraining the *lexical space* to literals which match a specific pattern. The value of *pattern* *must* be a *regular expression*.

*pattern* provides for:

- Constraining a *value space* to values that are denoted by literals which match a specific *regular expression*.

**Example**

The following is the definition of a *user–derived* datatype which is a better representation of postal codes in the United States, by limiting strings to those which are matched by a specific *regular expression*.

```xml
<simpleType name='better-us-zipcode'>
  <restriction base='string'>
    <pattern value='[0-9]{5}(-[0-9]{4})?'/>
  </restriction>
</simpleType>
```

### 4.3.4.1 The pattern Schema Component

**Schema Component: pattern**

{value} 

A *regular expression*.

{annotation}
4.3.4.2 XML Representation of pattern Schema Components

The XML representation for a pattern schema component is a `<pattern>` element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `pattern` Element Information Item

```xml
<pattern id = ID value = anySimpleType {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</pattern>
```

NOTE: The value must be a valid regular expression.

**pattern Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{value}</code></td>
<td>The actual value of the value {attribute}</td>
</tr>
<tr>
<td><code>{annotation}</code></td>
<td>The annotations corresponding to all the <code>&lt;annotation&gt;</code> element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

4.3.4.3 Constraints on XML Representation of pattern

Schema Representation Constraint: Multiple patterns

If multiple `<pattern>` element information items appear as [children] of a `<simpleType>`, the [value]s should be combined as if they appeared in a single regular expression, as separate branches.

NOTE: It is a consequence of the schema representation constraint Multiple patterns (§4.3.4.3) and of the rules for restriction: that pattern facets specified on the same step in a type derivation are ORed together, while pattern facets specified on different steps of a type derivation are ANDed together.

Thus, to impose two pattern constraints simultaneously, schema authors may either write a single pattern, which expresses the intersection of the two patterns they wish to impose, or define each pattern on a separate type derivation step.

4.3.4.4 pattern Validation Rules

Validation Rule: pattern valid

A literal in a lexical space is facet-valid with respect to pattern if: 1 the literal is among the set of character sequences denoted by the regular expression specified in `{value}`.

4.3.5 enumeration

[Definition:] enumeration constrains the value space to a specified set of values.
**enumeration** does not impose an order relation on the **value space** it creates; the value of the **ordered** property of the **derived** datatype remains that of the datatype from which it is **derived**.

**enumeration** provides for:

- Constraining a **value space** to a specified set of values.

Example The following example is a datatype definition for a **user-derived** datatype which limits the values of dates to the three US holidays enumerated. This datatype definition would appear in a schema authored by an "end-user" and shows how to define a datatype by enumerating the values in its **value space**. The enumerated values must be type-valid literals for the **base type**.

```xml
<simpleType name='holidays'>
  <annotation>
    <documentation>some US holidays</documentation>
  </annotation>
  <restriction base='gMonthDay'>
    <enumeration value='--01-01'>
      <annotation>
        <documentation>New Year's day</documentation>
      </annotation>
    </enumeration>
    <enumeration value='--07-04'>
      <annotation>
        <documentation>4th of July</documentation>
      </annotation>
    </enumeration>
    <enumeration value='--12-25'>
      <annotation>
        <documentation>Christmas</documentation>
      </annotation>
    </enumeration>
  </restriction>
</simpleType>
```

### 4.3.5.1 The enumeration Schema Component

**Schema Component: enumeration**

- **value**
  A set of values from the **value space** of the **base type definition**.
- **annotation**
  Optional. An **annotation**.

### 4.3.5.2 XML Representation of enumeration Schema Components

The XML representation for an **enumeration** schema component is an `<enumeration>` element information item. The correspondences between the properties of the information item and properties of the component are as follows:

**XML Representation Summary: enumeration Element Information Item**

```xml
<enumeration
  id = ID_
  value = anySimpleType
```

4.3 Constraining Facets
Content:  (annotation?)
</enumeration>

{value} ·must· be in the ·value space· of {base type definition}.

**enumeration Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value [attribute]</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

4.3.5.3 Constraints on XML Representation of enumeration

**Schema Representation Constraint: Multiple enumerations**
If multiple <enumeration> element information items appear as [children] of a <simpleType> the {value} of the enumeration component should be the set of all such [values].

4.3.5.4 enumeration Validation Rules

**Validation Rule: enumeration valid**
A value in a ·value space· is facet–valid with respect to ·enumeration· if the value is one of the values specified in {value}.

4.3.5.5 Constraints on enumeration Schema Components

**Schema Component Constraint: enumeration valid restriction**
It is an ·error· if any member of {value} is not in the ·value space· of {base type definition}.

4.3.6 whiteSpace

[Definition:]  **whiteSpace**  constrains the ·value space· of types ·derived· from string such that the various behaviors specified in Attribute Value Normalization in [XML 1.0 (Second Edition)] are realized. The value of whiteSpace must be one of {preserve, replace, collapse}.

- **preserve**
  
  No normalization is done, the value is not changed (this is the behavior required by [XML 1.0 (Second Edition)] for element content)

- **replace**
  
  All occurrences of #x9 (tab), #xA (line feed) and #xD (carriage return) are replaced with #x20 (space)

- **collapse**
  
  After the processing implied by replace, contiguous sequences of #x20's are collapsed to a single #x20, and leading and trailing #x20's are removed.

**NOTE:** The notation #xA used here (and elsewhere in this specification) represents the Universal Character Set (UCS) code point hexadecimal A (line feed), which is denoted by U+000A. This notation is to be distinguished from &xA;, which is the XML character reference to that same UCS code point.

4.3 Constraining Facets
**whiteSpace** is applicable to all **atomic** and **list** datatypes. For all **atomic** datatypes other than **string** (and types **derived** by **restriction** from it) the value of **whiteSpace** is **collapse** and cannot be changed by a schema author; for **string** the value of **whiteSpace** is **preserve**; for any type **derived** by **restriction** from **string** the value of **whiteSpace** can be any of the three legal values. For all datatypes **derived** by **list** the value of **whiteSpace** is **collapse** and cannot be changed by a schema author. For all datatypes **derived** by **union** **whiteSpace** does not apply directly; however, the normalization behavior of **union** types is controlled by the value of **whiteSpace** on that one of the **memberTypes** against which the **union** is successfully validated.

**NOTE:** For more information on **whiteSpace**, see the discussion on white space normalization in Schema Component Details in [XML Schema Part 1: Structures].

**whiteSpace** provides for:

- Constraining a **value space** according to the white space normalization rules.

Example The following example is the datatype definition for the **token** **built−in** **derived** datatype.

```xml
<simpleType name='token'>
  <restriction base='normalizedString'>
    <whiteSpace value='collapse'/>
  </restriction>
</simpleType>
```

### 4.3.6.1 The **whiteSpace** Schema Component

Schema Component: **whiteSpace**

- **value**
  - One of {preserve, replace, collapse}.
- **fixed**
  - A boolean.
- **annotation**
  - Optional. An annotation.

If **fixed** is **true**, then types for which the current type is the **base type definition** cannot specify a value for **whiteSpace** other than **value**.

### 4.3.6.2 XML Representation of **whiteSpace** Schema Components

The XML representation for a **whiteSpace** schema component is a **whiteSpace** element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: **whiteSpace** Element Information Item

```xml
<whiteSpace
  fixed = boolean : false
  id = ID
  value = (collapse | preserve | replace)
  {any attributes with non−schema namespace . . .}>
  {annotation?}
</whiteSpace>
```

4.3 Constraining Facets
whiteSpace Schema Component

Property                      Representation                      
{value}                      The actual value of the value [attribute].                      
{fixed}                      The actual value of the fixed [attribute], if present, otherwise false                      
{annotation}                The annotations corresponding to all the <annotation> element information items in the [children], if any.

4.3.6.3 whiteSpace Validation Rules

NOTE: There are no Validation Rules associated with whiteSpace. For more information, see the discussion on white space normalization in Schema Component Details in XML Schema Part 1: Structures.

4.3.6.4 Constraints on whiteSpace Schema Components

Schema Component Constraint: whiteSpace valid restriction
It is an error if whiteSpace is among the members of {facets} of {base type definition} and any of the following conditions is true: 1 {value} is replace or preserve and the {value} of the parent whiteSpace is collapse 2 {value} is preserve and the {value} of the parent whiteSpace is replace

4.3.7 maxInclusive

[Definition:] maxInclusive is the inclusive upper bound, of the value space, for a datatype with the ordered property. The value of maxInclusive must be in the value space of the base type.

maxInclusive provides for:

- Constraining a value space to values with a specific inclusive upper bound.

Example
The following is the definition of a user-derived datatype which limits values to integers less than or equal to 100, using maxInclusive.

<simpleType name='one-hundred-or-less'>
  <restriction base='integer'>
    <maxInclusive value='100'/>
  </restriction>
</simpleType>

4.3.7.1 The maxInclusive Schema Component

Schema Component: maxInclusive

{value}
A value from the value space of the base type definition.

{fixed}
A boolean.

{annotation}
Optional. An annotation.

4.3 Constraining Facets
If \( \text{fixed} \) is \textit{true}, then types for which the current type is the \( \text{base type definition} \) cannot specify a value for \( \text{maxInclusive} \) other than \( \{\text{value}\} \).

### 4.3.7.2 XML Representation of \textit{maxInclusive} Schema Components

The XML representation for a \( \text{maxInclusive} \) schema component is a \(<\text{maxInclusive}>\) element information item. The correspondences between the properties of the information item and properties of the component are as follows:

**XML Representation Summary:** \textit{maxInclusive} Element Information Item

\[
<\text{maxInclusive} \\
\quad \text{fixed} = \text{boolean}: \text{false} \\
\quad \text{id} = \text{ID} \\
\quad \text{value} = \text{anySimpleType} \\
\quad \{\text{any attributes with non-schema namespace . . .}\} > \\
\text{Content:} \quad \text{<annotation>?} \\
</\text{maxInclusive}>
\]

\( \{\text{value}\} \) \textit{must} be in the \textit{value space} of \( \{\text{base type definition}\} \).

**\textit{maxInclusive} Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {\text{value}} )</td>
<td>The actual value of the ( \text{value} ) ( {\text{attribute}} )</td>
</tr>
<tr>
<td>( {\text{fixed}} )</td>
<td>The actual value of the ( \text{fixed} ) ( {\text{attribute}} ), if present, otherwise false, if present, otherwise false</td>
</tr>
<tr>
<td>( {\text{annotation}} )</td>
<td>The annotations corresponding to all the (&lt;\text{annotation}&gt;) element information items in the {\text{children}}, if any.</td>
</tr>
</tbody>
</table>

### 4.3.7.3 maxInclusive Validation Rules

**Validation Rule:** \textit{maxInclusive Valid}

A value in an \textit{-ordered} \textit{-value space} is facet-valid with respect to \textit{-maxInclusive-}, determined as follows:1 if the \textit{-numeric} property in \{\text{fundamental facets}\} is \textit{true}, then the value \textit{must} be numerically less than or equal to \( \{\text{value}\} \);2 if the \textit{-numeric} property in \{\text{fundamental facets}\} is \textit{false} \( \{\text{i.e., base type definition} \) is one of the date and time related datatypes), then the value \textit{must} be chronologically less than or equal to \( \{\text{value}\} \).

### 4.3.7.4 Constraints on maxInclusive Schema Components

**Schema Component Constraint:** \textit{minInclusive} \( \leq \textit{maxInclusive} \)

It is an \textit{error} for the value specified for \textit{minInclusive} to be greater than the value specified for \textit{maxInclusive} for the same datatype. **Schema Component Constraint:** \textit{maxInclusive valid restriction}

It is an \textit{error} if any of the following conditions is true:1 \textit{maxInclusive} is among the members of \{\text{facets}\} of \{\text{base type definition}\} and \{\text{value}\} is greater than the \{\text{value}\} of the parent \textit{maxInclusive};2 \textit{maxExclusive} is among the members of \{\text{facets}\} of \{\text{base type definition}\} and \{\text{value}\} is greater than or equal to the \{\text{value}\} of the parent \textit{maxExclusive};3 \textit{minInclusive} is among the members of \{\text{facets}\} of \{\text{base type definition}\} and \{\text{value}\} is less than the \{\text{value}\} of the parent \textit{minInclusive};4 \textit{minExclusive} is among the members of \{\text{facets}\} of \{\text{base type definition}\} and \{\text{value}\} is less than or equal to the \{\text{value}\} of the parent \textit{minExclusive}
4.3.8 maxExclusive

[Definition:] **maxExclusive** is the **exclusive upper bound** of the **value space** for a datatype with the **ordered** property. The value of **maxExclusive** must be in the **value space** of the **base type**.

**maxExclusive** provides for:

- Constraining a **value space** to values with a specific **exclusive upper bound**.

Example The following is the definition of a **user-derived** datatype which limits values to integers less than or equal to 100, using **maxExclusive**.

```xml
<simpleType name='less-than-one-hundred-and-one'>
  <restriction base='integer'>
    <maxExclusive value='101'/>
  </restriction>
</simpleType>
```

Note that the **value space** of this datatype is identical to the previous one (named 'one-hundred-or-less').

4.3.8.1 The maxExclusive Schema Component

Schema Component: **maxExclusive**

{value}  
A value from the **value space** of the {base type definition}.

{fixed}  
A boolean.

{annotation}  
Optional. An annotation.

If {fixed} is true, then types for which the current type is the {base type definition} cannot specify a value for **maxExclusive** other than {value}.

4.3.8.2 XML Representation of maxExclusive Schema Components

The XML representation for a **maxExclusive** schema component is a `<maxExclusive>` element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: **maxExclusive** Element Information Item

```xml
<maxExclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}/>

Content:  (annotation?)
</maxExclusive>
```

{value} must be in the **value space** of {base type definition}.
maxExclusive Schema Component

Property Representation

{value} The actual value of the value [attribute].

{fixed} The actual value of the fixed [attribute], if present, otherwise false.

{annotation} The annotations corresponding to all the <annotation> element information items in the [children], if any.

4.3.8.3 maxExclusive Validation Rules

Validation Rule: maxExclusive Valid
A value in an _ordered_ ·value space· is facet−valid with respect to ·maxExclusive·, determined as follows:1 if the ·numeric· property in {fundamental facets} is true, then the value ·must· be numerically less than {value};2 if the ·numeric· property in {fundamental facets} is false (i.e., {base type definition} is one of the date and time related datatypes), then the value ·must· be chronologically less than {value};

4.3.8.4 Constraints on maxExclusive Schema Components

Schema Component Constraint: maxInclusive and maxExclusive
It is an ·error· for both ·maxInclusive· and ·maxExclusive· to be specified in the same derivation step of a datatype definition. Schema Component Constraint: minExclusive <= maxExclusive
It is an ·error· for the value specified for ·minExclusive· to be greater than the value specified for ·maxExclusive· for the same datatype. Schema Component Constraint: maxExclusive valid restriction
It is an ·error· if any of the following conditions is true:1 maxExclusive is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent maxExclusive2 maxInclusive is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent maxInclusive3 minInclusive is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of the parent minInclusive4 minExclusive is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of the parent minExclusive

4.3.9 minExclusive

[Definition:] minExclusive is the ·exclusive lower bound· of the ·value space· for a datatype with the ·ordered· property. The value of minExclusive ·must· be in the ·value space· of the ·base type·.

‑minExclusive· provides for:

− Constraining a ·value space· to values with a specific ·exclusive lower bound·.

Example The following is the definition of a ·user−derived· datatype which limits values to integers greater than or equal to 100, using ·minExclusive·.

<simpleType name='more-than-ninety-nine'>
  <restriction base='integer'>
    <minExclusive value='99' />
  </restriction>
</simpleType>

Note that the ·value space· of this datatype is identical to the previous one (named 'one−hundred−or−more').
4.3.9.1 The minExclusive Schema Component

Schema Component: minExclusive

{value}
A value from the -value space- of the {base type definition}.

{fixed}
A boolean.

{annotation}
Optional. An annotation.

If {fixed} is true, then types for which the current type is the {base type definition} cannot specify a value for minExclusive other than {value}.

4.3.9.2 XML Representation of minExclusive Schema Components

The XML representation for a minExclusive schema component is a <minExclusive> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: minExclusive Element Information Item

<minExclusive
 fixed = boolean : false
 id = ID
 value = anySimpleType
 {any attributes with non-schema namespace . . .}>
 Content: (annotation?)
</minExclusive>

{value} ·must· be in the ·value space· of {base type definition}.

minExclusive Schema Component

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value [attribute]</td>
</tr>
<tr>
<td>{fixed}</td>
<td>The actual value of the fixed [attribute], if present, otherwise false</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

4.3.9.3 minExclusive Validation Rules

Validation Rule: minExclusive Valid
A value in an -ordered- ·value space- is facet−valid with respect to -minExclusive- if:1 if the ·numeric−property in {fundamental facets} is true, then the value ·must· be numerically greater than {value};2 if the ·numeric−property in {fundamental facets} is false (i.e., {base type definition} is one of the date and time related datatypes), then the value ·must· be chronologically greater than {value};
4.3.9.4 Constraints on minExclusive Schema Components

Schema Component Constraint: minInclusive and minExclusive
It is an error for both minInclusive and minExclusive to be specified for the same datatype.

Schema Component Constraint: minExclusive < maxInclusive
It is an error for the value specified for minExclusive to be greater than or equal to the value specified for maxInclusive. minExclusive < maxInclusive is a valid restriction if any of the following conditions is true:
1. minExclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of the parent minExclusive.
2. maxInclusive is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent maxInclusive.
3. minInclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of the parent minInclusive.
4. maxExclusive is among the members of {facets} of {base type definition} and {value} is greater than or equal to the {value} of the parent maxExclusive.

4.3.10 minInclusive

[Definition:] minInclusive is the inclusive lower bound of the value space for a datatype with the ordered property. The value of minInclusive must be in the value space of the base type.

- minInclusive provides for:
  - Constraining a value space to values with a specific inclusive lower bound.

Example: The following is the definition of a user-derived datatype which limits values to integers greater than or equal to 100, using minInclusive.

```xml
<simpleType name='one-hundred-or-more'>
  <restriction base='integer'>
    <minInclusive value='100'/>
  </restriction>
</simpleType>
```

4.3.10.1 The minInclusive Schema Component

Schema Component: minInclusive

- **value**: A value from the value space of the base type definition.
- **fixed**: A boolean.
- **annotation**: Optional. An annotation.

If fixed is true, then types for which the current type is the base type definition cannot specify a value for minInclusive other than value.

4.3.10.2 XML Representation of minInclusive Schema Components

The XML representation for a minInclusive schema component is a <minInclusive> element information item. The correspondences between the properties of the information item and properties of the component are as follows:
XML Representation Summary: $\text{minInclusive}$ Element Information Item

\[
\begin{align*}
\text{<minInclusive} & \quad \text{fixed} = \text{boolean } : \text{false} \\
& \quad \text{id} = \text{ID} \\
& \quad \text{value} = \text{anySimpleType} \\
& \quad \text{(any attributes with non−schema namespace . . .)>
\end{align*}
\]

Content: (annotation?)

\[
\text{</minInclusive>}
\]

{value} ·must· be in the ·value space· of {base type definition}.

**minInclusive Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value {attribute}</td>
</tr>
<tr>
<td>{fixed}</td>
<td>The actual value of the fixed {attribute}, if present, otherwise false</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

4.3.10.3 minInclusive Validation Rules

**Validation Rule: minInclusive Valid**
A value in an -ordered ·value space- is facet−valid with respect to ·minInclusive· if:1 if the ·numeric· property in {fundamental facets} is true, then the value ·must· be numerically greater than or equal to {value};2 if the ·numeric· property in {fundamental facets} is false (i.e., {base type definition} is one of the date and time related datatypes), then the value ·must· be chronologically greater than or equal to {value};

4.3.10.4 Constraints on minInclusive Schema Components

**Schema Component Constraint: minInclusive < maxExclusive**
It is an ·error· for the value specified for ·minInclusive· to be greater than or equal to the value specified for ·maxExclusive· for the same datatype. **Schema Component Constraint: minInclusive valid restriction**
It is an ·error· if any of the following conditions is true:1 minInclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of the parent minInclusive2 maxInclusive is among the members of {facets} of {base type definition} and {value} is greater the {value} of the parent maxInclusive3 minExclusive is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of the parent minExclusive4 maxExclusive is among the members of {facets} of {base type definition} and {value} is greater than or equal to the {value} of the parent maxExclusive

4.3.11 totalDigits

[Definition:] **totalDigits** is the maximum number of digits in values of datatypes ·derived· from decimal. The value of **totalDigits** ·must· be a positiveInteger.

·totalDigits· provides for:

- Constraining a ·value space· to values with a specific maximum number of decimal digits (#x30–#x39).
Example The following is the definition of a user-derived datatype which could be used to represent monetary amounts, such as in a financial management application which does not have figures of $1M or more and only allows whole cents. This definition would appear in a schema authored by an "end-user" and shows how to define a datatype by specifying facet values which constrain the range of the base type in a manner specific to the base type (different than specifying max/min values as before).

```xml
<simpleType name='amount'>
  <restriction base='decimal'>
    <totalDigits value='8'/>
    <fractionDigits value='2' fixed='true'/>
  </restriction>
</simpleType>
```

4.3.11.1 The totalDigits Schema Component

Schema Component: totalDigits

{value}
A positiveInteger.

{fixed}
A boolean.

{annotation}
Optional. An annotation.

If {fixed} is true, then types for which the current type is the base type definition cannot specify a value for totalDigits other than {value}.

4.3.11.2 XML Representation of totalDigits Schema Components

The XML representation for a totalDigits schema component is a <totalDigits> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: totalDigits Element Information Item

```xml
<totalDigits
  fixed = boolean : false
  id = ID
  value = positiveInteger
  {any attributes with non-schema namespace . . .}>  
  Content:  (annotation?)
</totalDigits>
```

**totalDigits Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value {attribute}</td>
</tr>
<tr>
<td>{fixed}</td>
<td>The actual value of the fixed {attribute}, if present, otherwise false</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the [children], if any.</td>
</tr>
</tbody>
</table>
4.3.11.3 totalDigits Validation Rules

Validation Rule: totalDigits Valid
A value in a ·value space· is facet–valid with respect to totalDigits if: 1 the number of decimal digits in the value is less than or equal to {value};

4.3.11.4 Constraints on totalDigits Schema Components

Schema Component Constraint: totalDigits valid restriction
It is an ·error· if totalDigits is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent totalDigits.

4.3.12 fractionDigits

[Definition:] fractionDigits is the maximum number of digits in the fractional part of values of datatypes ·derived· from decimal. The value of fractionDigits ·must· be a nonNegativeInteger.

fractionDigits· provides for:

• Constraining a ·value space· to values with a specific maximum number of decimal digits in the fractional part.

Example The following is the definition of a ·user–derived· datatype which could be used to represent the magnitude of a person's body temperature on the Celsius scale. This definition would appear in a schema authored by an "end–user" and shows how to define a datatype by specifying facet values which constrain the range of the ·base type·.

```xml
<simpleType name='celsiusBodyTemp'>
  <restriction base='decimal'>
    <totalDigits value='4'/>
    <fractionDigits value='1'/>
    <minInclusive value='36.4'/>
    <maxInclusive value='40.5'/>
  </restriction>
</simpleType>
```

4.3.12.1 The fractionDigits Schema Component

Schema Component: fractionDigits

{value}  
A nonNegativeInteger.

{fixed}  
A boolean.

{annotation}  
Optional. An annotation.

If {fixed} is true, then types for which the current type is the {base type definition} cannot specify a value for fractionDigits other than {value}. 

4.3 Constraining Facets
4.3.12.2 XML Representation of fractionDigits Schema Components

The XML representation for a fractionDigits schema component is a <fractionDigits> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: fractionDigits Element Information Item

<fractionDigits
   fixed = boolean : false
   id = ID
   value = nonNegativeInteger
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?)
</fractionDigits>

fractionDigits Schema Component

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value {attribute}.</td>
</tr>
<tr>
<td>{fixed}</td>
<td>The actual value of the fixed {attribute}, if present, otherwise false</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

4.3.12.3 fractionDigits Validation Rules

Validation Rule: fractionDigits Valid
A value in a _value space_ is facet–valid with respect to _fractionDigits_, if: 1 the number of decimal digits in the fractional part of the value is less than or equal to _value_;

4.3.12.4 Constraints on fractionDigits Schema Components

Schema Component Constraint: fractionDigits less than or equal to totalDigits
It is an _error_ for _fractionDigits_ to be greater than _totalDigits_.

5 Conformance

This specification describes two levels of conformance for datatype processors. The first is required of all processors. Support for the other will depend on the application environments for which the processor is intended.

[Definition:] Minimally conforming processors must completely and correctly implement the Constraint on Schemas and Validation Rule.

[Definition:] Processors which accept schemas in the form of XML documents as described in XML Representation of Simple Type Definition Schema Components (§4.1.2) (and other relevant portions of Datatypes components (§4)) are additionally said to provide conformance to the XML Representation of Schemas, and must, when processing schema documents, completely and correctly implement all Schema Representation Constraint-s in this specification, and must adhere exactly to the specifications in XML Representation of Simple Type Definition Schema Components (§4.1.2) (and other relevant portions of
Datatype components (§4)) for mapping the contents of such documents to schema components for use in validation.

NOTE: By separating the conformance requirements relating to the concrete syntax of XML schema documents, this specification admits processors which validate using schemas stored in optimized binary representations, dynamically created schemas represented as programming language data structures, or implementations in which particular schemas are compiled into executable code such as C or Java. Such processors can be said to be minimally conforming but not necessarily in conformance to the XML Representation of Schemas.

A Schema for Datatype Definitions (normative)

```xml
<?xml version='1.0'?>
<!−− XML Schema schema for XML Schemas: Part 2: Datatypes −−>
<!DOCTYPE xs:schema PUBLIC "−//W3C//DTD XMLSCHEMA 200102//EN"
 "XMLSchema.dtd" [−−
keep this schema XML1.0 DTD valid−−>
<!ENTITY % schemaAttrs 'xmlns:hfp CDATA #IMPLIED'>
<!ELEMENT hfp:hasFacet EMPTY>
<!ATTLIST hfp:hasFacet name NMTOKEN #REQUIRED>
<!ELEMENT hfp:hasProperty EMPTY>
<!ATTLIST hfp:hasProperty name NMTOKEN #REQUIRED
value CDATA #REQUIRED>−−>
Make sure that processors that do not read the external subset will know about the various IDs we declare−−>
<!ATTLIST xs:simpleType id ID #IMPLIED>
<!ATTLIST xs:maxExclusive id ID #IMPLIED>
<!ATTLIST xs:minExclusive id ID #IMPLIED>
<!ATTLIST xs:maxInclusive id ID #IMPLIED>
<!ATTLIST xs:minInclusive id ID #IMPLIED>
<!ATTLIST xs:totalDigits id ID #IMPLIED>
<!ATTLIST xs:fractionDigits id ID #IMPLIED>
<!ATTLIST xs:length id ID #IMPLIED>
<!ATTLIST xs:minLength id ID #IMPLIED>
<!ATTLIST xs:maxLength id ID #IMPLIED>
<!ATTLIST xs:enumeration id ID #IMPLIED>
<!ATTLIST xs:pattern id ID #IMPLIED>
<!ATTLIST xs:appinfo id ID #IMPLIED>
<!ATTLIST xs:documentation id ID #IMPLIED>
<!ATTLIST xs:list id ID #IMPLIED>
<!ATTLIST xs:union id ID #IMPLIED>
]><xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3.org/2001/XMLSchema"
version="Id: datatypes.xsd,v 1.52 2001/04/27 11:49:21 ht Exp"
xmlns:hfp="http://www.w3.org/2001/XMLSchema−hasFacetAndProperty"
elementFormDefault="qualified" blockDefault="#all"/>
```

A Schema for Datatype Definitions (normative)
xml:lang="en">

<xs:annotation>
  <xs:documentation source="http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/datatypes">
    The schema corresponding to this document is normative, with respect to the syntactic constraints it expresses in the XML Schema language. The documentation (within &lt;documentation&gt; elements) below, is not normative, but rather highlights important aspects of the W3C Recommendation of which this is a part
  </xs:documentation>
</xs:annotation>

<xs:annotation>
  <xs:documentation>
    First the built-in primitive datatypes. These definitions are for information only, the real built-in definitions are magic. Note in particular that there is no type named 'anySimpleType'. The primitives should really be derived from no type at all, and anySimpleType should be derived as a union of all the primitives.
  </xs:documentation>
</xs:annotation>

<xs:documentation>
  For each built-in datatype in this schema (both primitive and derived) can be uniquely addressed via a URI constructed as follows:
  1) the base URI is the URI of the XML Schema namespace
  2) the fragment identifier is the name of the datatype

For example, to address the int datatype, the URI is:

  http://www.w3.org/2001/XMLSchema#int

Additionally, each facet definition element can be uniquely addressed via a URI constructed as follows:
  1) the base URI is the URI of the XML Schema namespace
  2) the fragment identifier is the name of the facet

For example, to address the maxInclusive facet, the URI is:

  http://www.w3.org/2001/XMLSchema#maxInclusive

Additionally, each facet usage in a built-in datatype definition can be uniquely addressed via a URI constructed as follows:
  1) the base URI is the URI of the XML Schema namespace
  2) the fragment identifier is the name of the datatype, followed by a period (".") followed by the name of the facet

For example, to address the usage of the maxInclusive facet in the definition of int, the URI is:

  http://www.w3.org/2001/XMLSchema#int.maxInclusive

</xs:documentation>
</xs:annotation>

<xs:simpleType name="string" id="string">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="length"/>
      <hfp:hasFacet name="minLength"/>
      <hfp:hasFacet name="maxLength"/>
    </xs:appinfo>
  </xs:annotation>
</xs:simpleType>
<xs:appinfo>
    <hfp:hasFacet name="pattern"/>
    <hfp:hasFacet name="enumeration"/>
    <hfp:hasFacet name="WhiteSpace"/>
    <hfp:hasFacet name="maxInclusive"/>
    <hfp:hasFacet name="maxExclusive"/>
    <hfp:hasFacet name="minInclusive"/>
    <hfp:hasFacet name="minExclusive"/>
    <hfp:hasProperty name="ordered" value="total"/>
    <hfp:hasProperty name="bounded" value="true"/>
    <hfp:hasProperty name="cardinality" value="finite"/>
    <hfp:hasProperty name="numeric" value="true"/>
</xs:appinfo>
<xs:documentation source="http://www.w3.org/TR/xmlschema-2/#double"/>
</xs:annotation>
<xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true"
        id="double.whiteSpace"/>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="decimal" id="decimal">
    <xs:annotation>
        <xs:appinfo>
            <hfp:hasFacet name="totalDigits"/>
            <hfp:hasFacet name="fractionDigits"/>
            <hfp:hasFacet name="pattern"/>
            <hfp:hasFacet name="WhiteSpace"/>
            <hfp:hasFacet name="enumeration"/>
            <hfp:hasFacet name="maxInclusive"/>
            <hfp:hasFacet name="maxExclusive"/>
            <hfp:hasFacet name="minInclusive"/>
            <hfp:hasFacet name="minExclusive"/>
            <hfp:hasProperty name="ordered" value="total"/>
            <hfp:hasProperty name="bounded" value="false"/>
            <hfp:hasProperty name="cardinality" value="countably infinite"/>
        </xs:appinfo>
        <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#decimal"/>
    </xs:annotation>
    <xs:restriction base="xs:anySimpleType">
        <xs:whiteSpace value="collapse" fixed="true"
            id="decimal.whiteSpace"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="duration" id="duration">
    <xs:annotation>
        <xs:appinfo>
            <hfp:hasFacet name="pattern"/>
            <hfp:hasFacet name="enumeration"/>
            <hfp:hasFacet name="WhiteSpace"/>
            <hfp:hasFacet name="maxInclusive"/>
            <hfp:hasFacet name="maxExclusive"/>
            <hfp:hasFacet name="minInclusive"/>
            <hfp:hasFacet name="minExclusive"/>
            <hfp:hasProperty name="ordered" value="partial"/>
            <hfp:hasProperty name="bounded" value="false"/>
        </xs:appinfo>
    </xs:annotation>
</xs:simpleType>
<xs:simpleType name="dateTime" id="dateTime">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="partial"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#dateTime"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">  
    <xs:whiteSpace value="collapse" fixed="true" id="dateTime.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="time" id="time">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="partial"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#time"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">  
    <xs:whiteSpace value="collapse" fixed="true" id="time.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="date" id="date">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="partial"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#date"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true" id="date.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="gYearMonth" id="gYearMonth">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="partial"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#gYearMonth"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true" id="gYearMonth.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="gYear" id="gYear">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="partial"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#gYear"/>
  </xs:annotation>
</xs:simpleType>
<xs:restriction base="xs:anySimpleType">
  <xs:whiteSpace value="collapse" fixed="true" id="gYear.whiteSpace"/>
</xs:restriction>
</xs:simpleType>

<xs:restriction base="xs:anySimpleType">
  <xs:whiteSpace value="collapse" fixed="true" id="gMonthDay.whiteSpace"/>
</xs:restriction>
</xs:simpleType>

<xs:restriction base="xs:anySimpleType">
  <xs:whiteSpace value="collapse" fixed="true" id="gDay.whiteSpace"/>
</xs:restriction>
</xs:simpleType>
A Schema for Datatype Definitions (normative)
NOTATION cannot be used directly in a schema; rather a type
must be derived from it by specifying at least one enumeration
facet whose value is the name of a NOTATION declared in the
schema.

Now the derived primitive types
<xs:simpleType name="language" id="language">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#language"/>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value="([a-zA-Z]{2}|[iI]-[a-zA-Z]+|[xX]-[a-zA-Z]{1,8})\((-[a-zA-Z]{1,8})\)*" id="language.pattern">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/REC-xml#NT-LanguageID">
          pattern specifies the content of section 2.12 of XML 1.0e2 and RFC 1766
        </xs:documentation>
      </xs:annotation>
    </xs:pattern>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="IDREFS" id="IDREFS">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="length"/>
      <hfp:hasFacet name="minLength"/>  
      <hfp:hasFacet name="maxLength"/>  
      <hfp:hasFacet name="enumeration"/> 
      <hfp:hasFacet name="whiteSpace"/> 
      <hfp:hasProperty name="ordered" value="false"/> 
      <hfp:hasProperty name="bounded" value="false"/>  
      <hfp:hasProperty name="cardinality" value="countably infinite"/>  
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#IDREFS"/>
  </xs:annotation>
  <xs:restriction>
    <xs:simpleType>
      <xs:list itemType="xs:IDREF"/>
    </xs:simpleType>
    <xs:minLength value="1" id="IDREFS.minLength"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ENTITIES" id="ENTITIES">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="length"/>
      <hfp:hasFacet name="minLength"/>
      <hfp:hasFacet name="maxLength"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasProperty name="ordered" value="false"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#ENTITIES"/>
  </xs:annotation>
  <xs:restriction>
    <xs:simpleType>
      <xs:list itemType="xs:ENTITY"/>
    </xs:simpleType>
    <xs:minLength value="1" id="ENTITIES.minLength"/>
  </xs:restriction>
</xs:simpleType>
XML Schema Part 0: Primer

A Schema for Datatype Definitions (normative) 300
pattern matches production 5 from the XML spec
</xs:documentation>
</xs:annotation>
</xs:pattern>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="NCName" id="NCName">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#NCName"/>
  </xs:annotation>
  <xs:restriction base="xs:Name">
    <xs:pattern value="\[i\-[:]]\[c\-[:]]*" id="NCName.pattern">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/REC-xml-names/#NT-NCName">
          pattern matches production 4 from the Namespaces in XML spec
        </xs:documentation>
      </xs:annotation>
    </xs:pattern>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ID" id="ID">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#ID"/>
  </xs:annotation>
  <xs:restriction base="xs:NCName"/>
</xs:simpleType>

<xs:simpleType name="IDREF" id="IDREF">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#IDREF"/>
  </xs:annotation>
  <xs:restriction base="xs:NCName"/>
</xs:simpleType>

<xs:simpleType name="ENTITY" id="ENTITY">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#ENTITY"/>
  </xs:annotation>
  <xs:restriction base="xs:NCName"/>
</xs:simpleType>

<xs:simpleType name="integer" id="integer">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#integer"/>
  </xs:annotation>
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="0" fixed="true" id="integer.fractionDigits"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="nonPositiveInteger" id="nonPositiveInteger">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#nonPositiveInteger"/>
  </xs:annotation>
</xs:simpleType>
<xs:simpleType name="nonPositiveInteger" id="nonPositiveInteger">
  <xs:annotation>
    <xs:documentation>
      source="http://www.w3.org/TR/xmlschema-2/#nonPositiveInteger" />
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:integer">
    <xs:maxInclusive value="0" id="nonPositiveInteger.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="negativeInteger" id="negativeInteger">
  <xs:annotation>
    <xs:documentation>
      source="http://www.w3.org/TR/xmlschema-2/#negativeInteger" />
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:nonPositiveInteger">
    <xs:maxInclusive value="-1" id="negativeInteger.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="long" id="long">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasProperty name="bounded" value="true"/>
      <hfp:hasProperty name="cardinality" value="finite"/>
    </xs:appinfo>
    <xs:documentation>
      source="http://www.w3.org/TR/xmlschema-2/#long" />
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="-9223372036854775808" id="long.minInclusive"/>
    <xs:maxInclusive value="9223372036854775807" id="long.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="int" id="int">
  <xs:annotation>
    <xs:documentation>
      source="http://www.w3.org/TR/xmlschema-2/#int" />
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:long">
    <xs:minInclusive value="-2147483648" id="int.minInclusive"/>
    <xs:maxInclusive value="2147483647" id="int.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="short" id="short">
  <xs:annotation>
    <xs:documentation>
      source="http://www.w3.org/TR/xmlschema-2/#short" />
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:int">
    <xs:minInclusive value="-32768" id="short.minInclusive"/>
    <xs:maxInclusive value="32767" id="short.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="byte" id="byte">
  <xs:annotation>
    <xs:documentation>
      source="http://www.w3.org/TR/xmlschema-2/#byte" />
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:short">
    <xs:minInclusive value="-128" id="byte.minInclusive"/>
  </xs:restriction>
</xs:simpleType>
<xs:maxInclusive value="127" id="byte.maxInclusive"/>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="nonNegativeInteger" id="nonNegativeInteger">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#nonNegativeInteger"/>
  </xs:annotation>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0" id="nonNegativeInteger.minInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="unsignedLong" id="unsignedLong">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasProperty name="bounded" value="true"/>
      <hfp:hasProperty name="cardinality" value="finite"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#unsignedLong"/>
  </xs:annotation>
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:maxInclusive value="18446744073709551615" id="unsignedLong.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="unsignedInt" id="unsignedInt">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#unsignedInt"/>
  </xs:annotation>
  <xs:restriction base="xs:unsignedLong">
    <xs:maxInclusive value="4294967295" id="unsignedInt.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="unsignedShort" id="unsignedShort">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#unsignedShort"/>
  </xs:annotation>
  <xs:restriction base="xs:unsignedInt">
    <xs:maxInclusive value="65535" id="unsignedShort.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="unsignedByte" id="unsignedBtype">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#unsignedByte"/>
  </xs:annotation>
  <xs:restriction base="xs:unsignedShort">
    <xs:maxInclusive value="255" id="unsignedByte.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="positiveInteger" id="positiveInteger">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#positiveInteger"/>
  </xs:annotation>
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:minInclusive value="1" id="positiveInteger.minInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="derivationControl">
  <xs:annotation>
    <xs:documentation>A utility type, not for public use</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="substitution"/>
    <xs:enumeration value="extension"/>
    <xs:enumeration value="restriction"/>
    <xs:enumeration value="list"/>
    <xs:enumeration value="union"/>
  </xs:restriction>
</xs:simpleType>

<xs:group name="simpleDerivation">
  <xs:choice>
    <xs:element ref="xs:restriction"/>
    <xs:element ref="xs:list"/>  
    <xs:element ref="xs:union"/>
  </xs:choice>
</xs:group>

<xs:simpleType name="simpleDerivationSet">
  <xs:annotation>
    <xs:documentation>#all or (possibly empty) subset of {restriction, union, list}
    A utility type, not for public use</xs:documentation>
  </xs:annotation>
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#all"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:derivationControl">
        <xs:enumeration value="list"/>
        <xs:enumeration value="union"/>
        <xs:enumeration value="restriction"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:complexType name="simpleType" abstract="true">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:group ref="xs:simpleDerivation"/>
      <xs:attribute name="final" type="xs:simpleDerivationSet"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:attribute name="name" type="xs:NCName">
  <xs:annotation>
    <xs:documentation>
      Can be restricted to required or forbidden
    </xs:documentation>
  </xs:annotation>
</xs:attribute>
</xs:extension>
</xs:complexType>

<xs:complexType name="topLevelSimpleType">
  <xs:complexContent>
    <xs:restriction base="xs:simpleType">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:simpleDerivation"/>
      </xs:sequence>
      <xs:attribute name="name" use="required"
        type="xs:NCName">
        <xs:annotation>
          <xs:documentation>
            Required at the top level
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="localSimpleType">
  <xs:complexContent>
    <xs:restriction base="xs:simpleType">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:simpleDerivation"/>
      </xs:sequence>
      <xs:attribute name="name" use="prohibited">
        <xs:annotation>
          <xs:documentation>
            Forbidden when nested
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="final" use="prohibited"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:element name="simpleType" type="xs:topLevelSimpleType" id="simpleType">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#element-simpleType"/>
  </xs:annotation>
</xs:element>

<xs:group name="facets">
  <xs:annotation>
    <xs:documentation>
      We should use a substitution group for facets, but
      that's ruled out because it would allow users to
add their own, which we're not ready for yet.

<xs:annotation>
  <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#element-restriction">
    base attribute and simpleType child are mutually exclusive, but one or other is required
  </xs:documentation>
</xs:annotation>

<xs:complexType>
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#element-list">
      itemType attribute and simpleType child are mutually exclusive, but one or other is required
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:sequence>
        <xs:element name="simpleType" type="xs:localSimpleType" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="itemType" type="xs:QName" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="union" id="union">
  <xs:complexType>
    <xs:annotation>
      <xs:documentation
        source="http://www.w3.org/TR/xmlschema-2/#element-union">
        memberTypes attribute must be non-empty or there must be
        at least one simpleType child
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:sequence>
          <xs:element name="simpleType" type="xs:localSimpleType"
            minOccurs="0" maxOccurs="unbounded"/>
          <xs:attribute name="memberTypes" use="optional">
            <xs:simpleType>
              <xs:list itemType="xs:QName"/>
            </xs:simpleType>
          </xs:attribute>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:complexType name="facet">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:attribute name="value" use="required"/>
      <xs:attribute name="fixed" type="xs:boolean" use="optional"
        default="false"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="noFixedFacet">
  <xs:complexContent>
    <xs:restriction base="xs:facet">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="fixed" use="prohibited"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:element name="minExclusive" id="minExclusive" type="xs:facet">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#element-minExclusive"/>
  </xs:annotation>
</xs:element>

<xs:element name="minInclusive" id="minInclusive" type="xs:facet">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#element-minInclusive"/>
  </xs:annotation>
</xs:element>
XML Schema Part 0: Primer

A Schema for Datatype Definitions (normative)
<xs:element name="maxLength" id="maxLength" type="xs:numFacet">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#element-maxLength"/>
  </xs:annotation>
</xs:element>

<xs:element name="enumeration" id="enumeration" type="xs:noFixedFacet">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#element-enumeration"/>
  </xs:annotation>
</xs:element>

<xs:element name="whiteSpace" id="whiteSpace">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#element-whiteSpace"/>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:restriction base="xs:facet">
        <xs:sequence>
          <xs:element ref="xs:annotation" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="value" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
              <xs:enumeration value="preserve"/>
              <xs:enumeration value="replace"/>
              <xs:enumeration value="collapse"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="pattern" id="pattern" type="xs:noFixedFacet">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#element-pattern"/>
  </xs:annotation>
</xs:element>

</xs:schema>

B DTD for Datatype Definitions (non–normative)

<!--
DTD for XML Schemas: Part 2: Datatypes
Id: datatypes.dtd,v 1.23 2001/03/16 17:36:30 ht Exp
Note this DTD is NOT normative, or even definitive.
-->

<!--
This DTD cannot be used on its own, it is intended
only for incorporation in XMLSchema.dtd, q.v.
-->
<!-- Define all the element names, with optional prefix -->
<!ENTITY % simpleType "%p;simpleType">
<!ENTITY % restriction "%p;restriction">
<!ENTITY % list "%p;list">
<!ENTITY % union "%p;union">
<!ENTITY % maxExclusive "%p;maxExclusive">
<!ENTITY % minExclusive "%p;minExclusive">
<!ENTITY % maxInclusive "%p;maxInclusive">
<!ENTITY % minInclusive "%p;minInclusive">
<!ENTITY % totalDigits "%p;totalDigits">
<!ENTITY % fractionDigits "%p;fractionDigits">
<!ENTITY % length "%p;length">
<!ENTITY % minLength "%p;minLength">
<!ENTITY % maxLength "%p;maxLength">
<!ENTITY % enumeration "%p;enumeration">
<!ENTITY % whiteSpace "%p;whiteSpace">
<!ENTITY % pattern "%p;pattern">

<!-- Customisation entities for the ATTLIST of each element type. Define one of these if your schema takes advantage of the anyAttribute='#other' in the schema for schemas -->
<!ENTITY % simpleTypeAttrs "">
<!ENTITY % restrictionAttrs "">
<!ENTITY % listAttrs "">
<!ENTITY % unionAttrs "">
<!ENTITY % maxExclusiveAttrs "">
<!ENTITY % minExclusiveAttrs "">
<!ENTITY % maxInclusiveAttrs "">
<!ENTITY % minInclusiveAttrs "">
<!ENTITY % totalDigitsAttrs "">
<!ENTITY % fractionDigitsAttrs "">
<!ENTITY % lengthAttrs "">
<!ENTITY % minLengthAttrs "">
<!ENTITY % maxLengthAttrs "">
<!ENTITY % enumerationAttrs "">
<!ENTITY % whiteSpaceAttrs "">
<!ENTITY % patternAttrs "">

<!-- Define some entities for informative use as attribute types -->
<!ENTITY % URIref "CDATA">
<!ENTITY % XPathExpr "CDATA">
<!ENTITY % QName "NMTOKEN">
<!ENTITY % QNames "NMTOKENS">
<!ENTITY % NCName "NMTOKEN">
<!ENTITY % nonNegativeInteger "NMTOKEN">
<!ENTITY % boolean "(true|false)">
<!ENTITY % simpleDerivationSet "CDATA">

<!-- Note that the use of 'facet' below is less restrictive than is really intended: There should in fact be no more than one of each of minInclusive, minExclusive, maxInclusive, maxExclusive, totalDigits, fractionDigits,
length, maxLength, minLength within datatype, and the min- and max- variants of Inclusive and Exclusive are mutually exclusive. On the other hand, pattern and enumeration may repeat.

-->
<!ENTITY % minBound "(%minInclusive; | %minExclusive;)">
<!ENTITY % maxBound "(%maxInclusive; | %maxExclusive;)">
<!ENTITY % bounds "(%minBound; | %maxBound;)">
<!ENTITY % numeric "(%totalDigits; | %fractionDigits;)">
<!ENTITY % unordered "%pattern; | %enumeration; | %whiteSpace; | %length; | %maxLength; | %minLength;">  
<!ENTITY % facet "%ordered; | %unordered;">  
<!ENTITY % facetAttr "value CDATA #REQUIRED
    id ID #IMPLIED">
<!ENTITY % fixedAttr "fixed %boolean; #IMPLIED">
<!ENTITY % facetModel "(%annotation;)?">
<!ELEMENT %simpleType;  
   ((%annotation;)?, (%restriction; | %list; | %union;))>
<!ATTLIST %simpleType;  
   name %NCName; #IMPLIED
  final %simpleDerivationSet; #IMPLIED
  id ID #IMPLIED
  %simpleTypeAttrs;>
<!-- name is required at top level -->
<!ELEMENT %restriction; ((%annotation;)?,
   (%restriction1; | (%simpleType;)?,(%facet;)*)),
   (%attrDecls;))>
<!ATTLIST %restriction;  
   base %QName;                  #IMPLIED
   id ID       #IMPLIED
   %restrictionAttrs;>
<!-- base and simpleType child are mutually exclusive, one is required. -->
   restriction is shared between simpleType and
   simpleContent and complexContent (in XMLSchema.xsd).
   restriction1 is for the latter cases, when this
   is restricting a complex type, as is attrDecls.
-->
<!ELEMENT %list; ((%annotation;)?,(%simpleType;)?)>  
<!ATTLIST %list;  
   itemType %QName;             #IMPLIED
   id ID       #IMPLIED
   %listAttrs;>
<!-- itemType and simpleType child are mutually exclusive, one is required -->
<!ELEMENT %union; ((%annotation;)?,(%simpleType;)*)>  
<!ATTLIST %union;  
   id ID       #IMPLIED
   memberTypes %QNames;            #IMPLIED
   %unionAttrs;>
<!-- At least one item in memberTypes or one simpleType child is required -->
-->

<!ELEMENT %maxExclusive; %facetModel;>
<!ATTLIST %maxExclusive; %facetAttr; %fixedAttr; %maxExclusiveAttrs;>

<!ELEMENT %minExclusive; %facetModel;>
<!ATTLIST %minExclusive; %facetAttr; %fixedAttr; %minExclusiveAttrs;>

<!ELEMENT %maxInclusive; %facetModel;>
<!ATTLIST %maxInclusive; %facetAttr; %fixedAttr; %maxInclusiveAttrs;>

<!ELEMENT %minInclusive; %facetModel;>
<!ATTLIST %minInclusive; %facetAttr; %fixedAttr; %minInclusiveAttrs;>

<!ELEMENT %totalDigits; %facetModel;>
<!ATTLIST %totalDigits; %facetAttr; %fixedAttr; %totalDigitsAttrs;>

<!ELEMENT %fractionDigits; %facetModel;>
<!ATTLIST %fractionDigits; %facetAttr; %fixedAttr; %fractionDigitsAttrs;>

<!ELEMENT %length; %facetModel;>
<!ATTLIST %length; %facetAttr; %fixedAttr; %lengthAttrs;>

<!ELEMENT %minLength; %facetModel;>
<!ATTLIST %minLength; %facetAttr; %fixedAttr; %minLengthAttrs;>

<!ELEMENT %maxLength; %facetModel;>
<!ATTLIST %maxLength; %facetAttr; %fixedAttr; %maxLengthAttrs;>

<!-- This one can be repeated -->
<!ELEMENT %enumeration; %facetModel;>
<!ATTLIST %enumeration; %facetAttr; %enumerationAttrs;>

<!ELEMENT %whiteSpace; %facetModel;>
<!ATTLIST %whiteSpace; %facetAttr; %fixedAttr;>
### C Datatypes and Facets

#### C.1 Fundamental Facets

The following table shows the values of the fundamental facets for each built-in datatype.

<table>
<thead>
<tr>
<th>Datatype</th>
<th>ordered</th>
<th>bounded</th>
<th>cardinality</th>
<th>numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
<td>false</td>
<td>finite</td>
<td>false</td>
</tr>
<tr>
<td>float</td>
<td>total</td>
<td>true</td>
<td>finite</td>
<td>true</td>
</tr>
<tr>
<td>double</td>
<td>total</td>
<td>true</td>
<td>finite</td>
<td>true</td>
</tr>
<tr>
<td>decimal</td>
<td>total</td>
<td>false</td>
<td>countably infinite</td>
<td>true</td>
</tr>
<tr>
<td>duration</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>dateTime</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>time</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>date</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gYear</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gDay</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gMonth</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>hexBinary</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>base64Binary</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>anyURI</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>QName</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>NOTATION</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>normalizedString</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>token</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>language</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>IDREFS</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>NM_TOKEN</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>NM_TOKENS</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>Name</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
</tbody>
</table>
D ISO 8601 Date and Time Formats

D.1 ISO 8601 Conventions

The primitive datatypes `duration`, `dateTime`, `time`, `date`, `gYearMonth`, `gMonthDay`, `gDay`, `gMonth` and `gYear` use lexical formats inspired by [ISO 8601]. This appendix provides more detail on the ISO formats and discusses some deviations from them for the datatypes defined in this specification.

[ISO 8601] "specifies the representation of dates in the proleptic Gregorian calendar and times and representations of periods of time". The proleptic Gregorian calendar includes dates prior to 1582 (the year it came into use as an ecclesiastical calendar). It should be pointed out that the datatypes described in this specification do not cover all the types of data covered by [ISO 8601], nor do they support all the lexical representations for those types of data.

[ISO 8601] lexical formats are described using "pictures" in which characters are used in place of digits. For the primitive datatypes `dateTime`, `time`, `date`, `gYearMonth`, `gMonthDay`, `gDay`, `gMonth` and `gYear`, these characters have the following meanings:

- C --- represents a digit used in the thousands and hundreds components, the "century" component, of the time element "year". Legal values are from 0 to 9.
- Y --- represents a digit used in the tens and units components of the time element "year". Legal values are from 0 to 9.
- M --- represents a digit used in the time element "month". The two digits in a MM format can have values from 1 to 12.
- D --- represents a digit used in the time element "day". The two digits in a DD format can have values from 1 to 28 if the month value equals 2, 1 to 29 if the month value equals 2 and the year is a leap year, 1 to 30 if the month value equals 4, 6, 9 or 11, and 1 to 31 if the month value equals 1, 3, 5, 7, 8, 10 or 12.
• h — represents a digit used in the time element "hour". The two digits in a hh format can have values from 0 to 23.
• m — represents a digit used in the time element "minute". The two digits in a mm format can have values from 0 to 59.
• s — represents a digit used in the time element "second". The two digits in a ss format can have values from 0 to 60. In the formats described in this specification the whole number of seconds may be followed by decimal seconds to an arbitrary level of precision. This is represented in the picture by "ss.sss". A value of 60 or more is allowed only in the case of leap seconds.

Strictly speaking, a value of 60 or more is not sensible unless the month and day could represent March 31, June 30, September 30, or December 31 in UTC. Because the leap second is added or subtracted as the last second of the day in UTC time, the long (or short) minute could occur at other times in local time. In cases where the leap second is used with an inappropriate month and day it, and any fractional seconds, should be considered as added or subtracted from the following minute.

For all the information items indicated by the above characters, leading zeros are required where indicated.

In addition to the above, certain characters are used as designators and appear as themselves in lexical formats.

• T — is used as time designator to indicate the start of the representation of the time of day in dateTime.
• Z — is used as time−zone designator, immediately (without a space) following a data element expressing the time of day in Coordinated Universal Time (UTC) in dateTime, time, date, gYearMonth, gMonthDay, gDay, gMonth, and gYear.

In the lexical format for duration the following characters are also used as designators and appear as themselves in lexical formats:

• P — is used as the time duration designator, preceding a data element representing a given duration of time.
• Y — follows the number of years in a time duration.
• M — follows the number of months or minutes in a time duration.
• D — follows the number of days in a time duration.
• H — follows the number of hours in a time duration.
• S — follows the number of seconds in a time duration.

The values of the Year, Month, Day, Hour and Minutes components are not restricted but allow an arbitrary integer. Similarly, the value of the Seconds component allows an arbitrary decimal. Thus, the lexical format for duration and datatypes derived from it does not follow the alternative format of § 5.5.3.2.1 of [ISO 8601].

D.2 Truncated and Reduced Formats

[ISO 8601] supports a variety of "truncated" formats in which some of the characters on the left of specific formats, for example, the century, can be omitted. Truncated formats are, in general, not permitted for the datatypes defined in this specification with three exceptions. The time datatype uses a truncated format for dateTime which represents an instant of time that recurs every day. Similarly, the gMonthDay and gDay datatypes use left−truncated formats for date. The datatype gMonth uses a right and left truncated format for date.
ISO 8601 also supports a variety of "reduced" or right-truncated formats in which some of the characters to the right of specific formats, such as the time specification, can be omitted. Right truncated formats are also, in general, not permitted for the datatypes defined in this specification with the following exceptions: right-truncated representations of dateTime are used as lexical representations for date, gMonth, gYear.

D.3 Deviations from ISO 8601 Formats

D.3.1 Sign Allowed

An optional minus sign is allowed immediately preceding, without a space, the lexical representations for duration, dateTime, date, gMonth, gYear.

D.3.2 No Year Zero

The year "0000" is an illegal year value.

D.3.3 More Than 9999 Years

To accommodate year values greater than 9999, more than four digits are allowed in the year representations of dateTime, date, gYearMonth, and gYear. This follows [ISO 8601 Draft Revision].

E Adding durations to dateTimes

Given a dateTime S and a duration D, this appendix specifies how to compute a dateTime E where E is the end of the time period with start S and duration D i.e. E = S + D. Such computations are used, for example, to determine whether a dateTime is within a specific time period. This appendix also addresses the addition of durations to the datatypes date, gYearMonth, gYear, gDay and gMonth, which can be viewed as a set of dateTimes. In such cases, the addition is made to the first or starting dateTime in the set.

This is a logical explanation of the process. Actual implementations are free to optimize as long as they produce the same results. The calculation uses the notation S[year] to represent the year field of S, S[month] to represent the month field, and so on. It also depends on the following functions:

- fQuotient(a, b) = the greatest integer less than or equal to a/b
  - fQuotient(−1, 3) = −1
  - fQuotient(0, 3) ... fQuotient(2, 3) = 0
  - fQuotient(3, 3) = 1
  - fQuotient(3.123, 3) = 1

- modulo(a, b) = a − fQuotient(a,b)*b
  - modulo(−1, 3) = 2
  - modulo(0, 3) ... modulo(2, 3) = 0 ... 2
  - modulo(3, 3) = 0
  - modulo(3.123, 3) = 0.123

- fQuotient(a, low, high) = fQuotient(a − low, high − low)
  - fQuotient(0, 1, 13) = −1
  - fQuotient(1, 1, 13) ... fQuotient(12, 1, 13) = 0
  - fQuotient(13, 1, 13) = 1
  - fQuotient(13.123, 1, 13) = 1
• modulo(a, low, high) = modulo(a − low, high − low) + low
  ♦ modulo(0, 1, 13) = 12
  ♦ modulo(1, 1, 13) = 1
  ♦ modulo(13.123, 1, 13) = 1.123
• maximumDayInMonthFor(yearValue, monthValue) =
  ♦ M := modulo(monthValue, 1, 13)
  ♦ Y := yearValue + fQuotient(monthValue, 1, 13)
  ♦ Return a value based on M and Y:

<table>
<thead>
<tr>
<th>M</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>January, March, May, July, August, October, or December</td>
</tr>
<tr>
<td>30</td>
<td>April, June, September, or November</td>
</tr>
<tr>
<td>29</td>
<td>February AND (modulo(Y, 400) = 0 OR (modulo(Y, 100) != 0) AND modulo(Y, 4) = 0)</td>
</tr>
<tr>
<td>28</td>
<td>Otherwise</td>
</tr>
</tbody>
</table>

**E.1 Algorithm**

Essentially, this calculation is equivalent to separating D into <year,month> and <day,hour,minute,second> fields. The <year,month> is added to S. If the day is out of range, it is pinned to be within range. Thus April 31 turns into April 30. Then the <day,hour,minute,second> is added. This latter addition can cause the year and month to change.

Leap seconds are handled by the computation by treating them as overflows. Essentially, a value of 60 seconds in S is treated as if it were a duration of 60 seconds added to S (with a zero seconds field). All calculations thereafter use 60 seconds per minute.

Thus the addition of either PT1M or PT60S to any dateTime will always produce the same result. This is a special definition of addition which is designed to match common practice, and — most importantly — be stable over time.

A definition that attempted to take leap−seconds into account would need to be constantly updated, and could not predict the results of future implementation's additions. The decision to introduce a leap second in UTC is the responsibility of the [International Earth Rotation Service (IERS)]. They make periodic announcements as to when leap seconds are to be added, but this is not known more than a year in advance. For more information on leap seconds, see [U.S. Naval Observatory Time Service Department].

The following is the precise specification. These steps must be followed in the same order. If a field in D is not specified, it is treated as if it were zero. If a field in S is not specified, it is treated in the calculation as if it were the minimum allowed value in that field, however, after the calculation is concluded, the corresponding field in E is removed (set to unspecified).

• Months (may be modified additionally below)
  ♦ temp := S[month] + D[month]
  ♦ E[month] := modulo(temp, 1, 13)
  ♦ carry := fQuotient(temp, 1, 13)
• **Years (may be modified additionally below)**
  ◦ \( E[\text{year}] := S[\text{year}] + D[\text{year}] + \text{carry} \)

• **Zone**
  ◦ \( E[\text{zone}] := S[\text{zone}] \)

• **Seconds**
  ◦ \( \text{temp} := S[\text{second}] + D[\text{second}] \)
  ◦ \( E[\text{second}] := \text{modulo}(\text{temp}, 60) \)
  ◦ \( \text{carry} := \text{fQuotient}(\text{temp}, 60) \)

• **Minutes**
  ◦ \( \text{temp} := S[\text{minute}] + D[\text{minute}] + \text{carry} \)
  ◦ \( E[\text{minute}] := \text{modulo}(\text{temp}, 60) \)
  ◦ \( \text{carry} := \text{fQuotient}(\text{temp}, 60) \)

• **Hours**
  ◦ \( \text{temp} := S[\text{hour}] + D[\text{hour}] + \text{carry} \)
  ◦ \( E[\text{hour}] := \text{modulo}(\text{temp}, 24) \)
  ◦ \( \text{carry} := \text{fQuotient}(\text{temp}, 24) \)

• **Days**
  ◦ if \( S[\text{day}] > \text{maximumDayInMonthFor}(E[\text{year}], E[\text{month}]) \)
    ◦ \( \text{tempDays} := \text{maximumDayInMonthFor}(E[\text{year}], E[\text{month}]) \)
  ◦ else if \( S[\text{day}] < 1 \)
    ◦ \( \text{tempDays} := 1 \)
  ◦ else
    ◦ \( \text{tempDays} := S[\text{day}] \)
  ◦ \( E[\text{day}] := \text{tempDays} + D[\text{day}] + \text{carry} \)
  ◦ **START LOOP**
    ◦ **IF** \( E[\text{day}] < 1 \)
      ◦ \( E[\text{day}] := E[\text{day}] + \text{maximumDayInMonthFor}(E[\text{year}], E[\text{month}] - 1) \)
      ◦ \( \text{carry} := -1 \)
    ◦ **ELSE IF** \( E[\text{day}] > \text{maximumDayInMonthFor}(E[\text{year}], E[\text{month}]) \)
      ◦ \( E[\text{day}] := E[\text{day}] - \text{maximumDayInMonthFor}(E[\text{year}], E[\text{month}]) \)
      ◦ \( \text{carry} := 1 \)
    ◦ **ELSE** **EXIT LOOP**
    ◦ \( \text{temp} := E[\text{month}] + \text{carry} \)
    ◦ \( E[\text{month}] := \text{modulo}(\text{temp}, 1, 13) \)
    ◦ \( E[\text{year}] := E[\text{year}] + \text{fQuotient}(\text{temp}, 1, 13) \)
    ◦ **GOTO START LOOP**

**Examples:**

<table>
<thead>
<tr>
<th>dateTime</th>
<th>duration</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000–01</td>
<td>−P3M</td>
<td>1999–10</td>
</tr>
<tr>
<td>2000–01–12</td>
<td>PT33H</td>
<td>2000–01–13</td>
</tr>
</tbody>
</table>
E.2 Commutativity and Associativity

Time durations are added by simply adding each of their fields, respectively, without overflow.

The order of addition of durations to instants is significant. For example, there are cases where:

\[((\text{dateTime} + \text{duration1}) + \text{duration2}) \neq ((\text{dateTime} + \text{duration2}) + \text{duration1})\]

*Example:*

\[(2000−03−30 + \text{P1D}) + \text{P1M} = 2000−03−31 + \text{P1M} = 2001−04−30\]

\[(2000−03−30 + \text{P1M}) + \text{P1D} = 2000−04−30 + \text{P1D} = 2000−05−01\]

F Regular Expressions

A regular expression, \( R \) is a sequence of characters that denote a set of strings \( L(R) \). When used to constrain a lexical space, a regular expression \( R \) asserts that only strings in \( L(R) \) are valid literals for values of that type.

[Definition:] A regular expression is composed from zero or more branch-es, separated by | characters.

<table>
<thead>
<tr>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] ( \text{regExp ::= branch \ ( '</td>
</tr>
</tbody>
</table>

[Definition:] A branch consists of zero or more piece-s, concatenated together.

<table>
<thead>
<tr>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2] ( \text{branch ::= piece*} )</td>
</tr>
</tbody>
</table>

[Definition:] A piece is an atom-, possibly followed by a quantifier.

<table>
<thead>
<tr>
<th>Piece</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3] ( \text{piece ::= atom \ quantifier?} )</td>
</tr>
</tbody>
</table>

**NOTE:** The regular expression language in the Perl Programming Language [Perl] does not include a quantifier of the form \( S \{ , \ m \} \), since it is logically equivalent to \( S \{ 0, m \} \). We have, therefore, left this logical possibility out of the regular expression language defined by this specification. We welcome further input from implementors and schema authors on this issue.

[Definition:] A quantifier is one of ?, *, +, \{n, m\} or \{n, \}, which have the meanings defined in the table above.
Quanitifier

[4] quantifier ::= [?*+] | ( '(' quantity ')' )
[5] quantity ::= quantRange | quantMin | QuantExact
[6] quantRange ::= QuantExact ',' QuantExact
[7] quantMin ::= QuantExact ','
[8] QuantExact ::= [0−9]+

[Definition:] An atom is either a normal character, a character class, or a parenthesized regular expression.

Atom

[9] atom ::= Char | charClass | ( '(' regExp ')' )

[Definition:] A metacharacter is either ., \, ?, *, +, (, ), [, or ]. These characters have special meanings in regular expressions, but can be escaped to form atom-s that denote the sets of strings containing only themselves, i.e., an escaped metacharacter behaves like a normal character.

[Definition:] A normal character is any XML character that is not a metacharacter. In regular expressions, a normal character is an atom that denotes the singleton set of strings containing only itself.

Normal Character

[10] Char ::= [^\.?*+()\#x5B\#x5D]

Note that a normal character can be represented either as itself, or with a character reference.

F.1 Character Classes

[Definition:] A character class is an atom R that identifies a set of characters C(R). The set of strings L(R) denoted by a character class R contains one single-character string "c" for each character c in C(R).

Character Class

[11] charClass ::= charClassEsc | charClassExpr

A character class is either a character class escape or a character class expression.

[Definition:] A character class expression is a character group surrounded by [ and ] characters. For all character groups G, [G] is a valid character class expression, identifying the set of characters C([G]) = C(G).
A character group is either a positive character group, a negative character group, or a character class subtraction.

A positive character group consists of one or more character range-s or character class escape-s, concatenated together. A positive character group identifies the set of characters containing all of the characters in all of the sets identified by its constituent ranges or escapes.

A negative character group is a positive character group preceded by the ^ character. For all positive character group-s P, ^P is a valid negative character group, and C(^P) contains all XML characters that are not in C(P).

A character class subtraction is a character class expression subtracted from a positive character group or negative character group, using the − character.

For any positive character group, or negative character group, G, and any character class expression, C, G−C is a valid character class subtraction, identifying the set of all characters in C(G) that are not also in C(C).

A character range R identifies a set of characters C(R) containing all XML characters with UCS code points in a specified range.
A single XML character is a ·character range· that identifies the set of characters containing only itself. All
XML characters are valid character ranges, except as follows:

- The [, ], and \ characters are not valid character ranges;
- The ^ character is only valid at the beginning of a ·positive character group· if it is part of a ·negative
  character group·; and
- The – character is a valid character range only at the beginning or end of a ·positive character group·.

A ·character range· ·may· also be written in the form s−e, identifying the set that contains all XML characters
with UCS code points greater than or equal to the code point of s, but not greater than the code point of e.

\( s−e \) is a valid character range iff:

- \( s \) is a ·single character escape·, or an XML character;
- \( s \) is not \( \backslash \);
- If \( s \) is the first character in a ·character class expression·, then \( s \) is not ^
- \( e \) is a ·single character escape·, or an XML character;
- \( e \) is not \( \backslash \) or [; and
- The code point of \( e \) is greater than or equal to the code point of \( s \);

**NOTE:** The code point of a ·single character escape· is the code point of the single character
in the set of characters that it identifies.

### F.1.1 Character Class Escapes

[Definition:] A ·character class escape· is a short sequence of characters that identifies predefined
character class. The valid character class escapes are the ·single character escape·s, the ·multi−character
escape·s, and the ·category escape·s (including the ·block escape·s).

#### Character Class Escape

| [23] charClassEsc | ::= | ( SingleCharEsc | MultiCharEsc | catEsc | complEsc ) |

[Definition:] A ·single character escape· identifies a set containing a only one character — usually
because that character is difficult or impossible to write directly into a ·regular expression·.
Single Character Escape

\[24\] \text{SingleCharEsc ::= '\ \ [nrt]\ | .?*+()\{\}_#x2D#x5B#x5D#x5E]}

[Definition:] [Unicode Database] specifies a number of possible values for the "General Category" property and provides mappings from code points to specific character properties. The set containing all characters that have property \(X\), can be identified with a category escape \(\p{X}\). The complement of this set is specified with the category escape \(\P{X}\). (\(\P{X}\) = \(\{\ ^{\p{X}}\}\)).

Category Escape

\[25\] \text{catEsc ::= 'p{' charProp '}}\)
\[26\] \text{complEsc ::= 'P{' charProp '}}\)
\[27\] \text{charProp ::= IsCategory | IsBlock}

NOTE: [Unicode Database] is subject to future revision. For example, the mapping from code points to character properties might be updated. All minimally conforming processors must support the character properties defined in the version of [Unicode Database] that is current at the time this specification became a W3C Recommendation. However, implementors are encouraged to support the character properties defined in any future version.

The following table specifies the recognized values of the "General Category" property.

<table>
<thead>
<tr>
<th>Category</th>
<th>Property</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letters</td>
<td>L</td>
<td>All Letters</td>
</tr>
<tr>
<td></td>
<td>Lu</td>
<td>uppercase</td>
</tr>
<tr>
<td></td>
<td>Ll</td>
<td>lowercase</td>
</tr>
<tr>
<td></td>
<td>Lt</td>
<td>titlecase</td>
</tr>
<tr>
<td></td>
<td>Lm</td>
<td>modifier</td>
</tr>
<tr>
<td></td>
<td>Lo</td>
<td>other</td>
</tr>
<tr>
<td>Marks</td>
<td>M</td>
<td>All Marks</td>
</tr>
<tr>
<td></td>
<td>Mn</td>
<td>nonspacing</td>
</tr>
<tr>
<td></td>
<td>Mc</td>
<td>spacing combining</td>
</tr>
</tbody>
</table>
## F.1 Character Classes

### Me
- enclosing

### Numbers
- **N**: All Numbers
- **Nd**: decimal digit
- **Nl**: letter
- **No**: other

### Punctuation
- **P**: All Punctuation
- **Pc**: connector
- **Pd**: dash
- **Ps**: open
- **Pe**: close
- **Pi**: initial quote (may behave like Ps or Pe depending on usage)
- **Pf**: final quote (may behave like Ps or Pe depending on usage)
- **Po**: other

### Separators
- **Z**: All Separators
- **Zs**: space
- **Zl**: line
- **Zp**: paragraph

### Symbols
- **S**: All Symbols
<table>
<thead>
<tr>
<th></th>
<th>Sm</th>
<th>math</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sc</td>
<td>currency</td>
</tr>
<tr>
<td></td>
<td>Sk</td>
<td>modifier</td>
</tr>
<tr>
<td></td>
<td>So</td>
<td>other</td>
</tr>
</tbody>
</table>

### Other

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>All Others</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cc</td>
<td>control</td>
</tr>
<tr>
<td></td>
<td>Cf</td>
<td>format</td>
</tr>
<tr>
<td></td>
<td>Co</td>
<td>private use</td>
</tr>
<tr>
<td></td>
<td>Cn</td>
<td>not assigned</td>
</tr>
</tbody>
</table>

### Categories

<table>
<thead>
<tr>
<th></th>
<th>IsCategory</th>
<th>::=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Letters</td>
<td>Marks</td>
</tr>
<tr>
<td>[28]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** The properties mentioned above exclude the Cs property. The Cs property identifies "surrogate" characters, which do not occur at the level of the "character abstraction" that XML instance documents operate on.

[Definition:] [Unicode Database] groups code points into a number of blocks such as Basic Latin (i.e., ASCII), Latin−1 Supplement, Hangul Jamo, CJK Compatibility, etc. The set containing all characters that have block name X (with all white space stripped out), can be identified with a block escape \p{IsX}. The complement of this set is specified with the block escape \P{IsX}.([\p{IsX}] = [^\p{IsX}]).
The following table specifies the recognized block names (for more information, see the "Blocks.txt" file in [Unicode Database]).

<table>
<thead>
<tr>
<th>Start Code</th>
<th>End Code</th>
<th>Block Name</th>
<th>Start Code</th>
<th>End Code</th>
<th>Block Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>#x0000</td>
<td>#x007F</td>
<td>BasicLatin</td>
<td>#x0080</td>
<td>#x00FF</td>
<td>Latin–1Supplement</td>
</tr>
<tr>
<td>#x0100</td>
<td>#x017F</td>
<td>LatinExtended–A</td>
<td>#x0180</td>
<td>#x024F</td>
<td>LatinExtended–B</td>
</tr>
<tr>
<td>#x0250</td>
<td>#x02AF</td>
<td>IPAExtensions</td>
<td>#x02B0</td>
<td>#x02FF</td>
<td>SpacingModifierLetters</td>
</tr>
<tr>
<td>#x0300</td>
<td>#x036F</td>
<td>CombiningDiacriticalMarks</td>
<td>#x0370</td>
<td>#x03FF</td>
<td>Greek</td>
</tr>
<tr>
<td>#x0400</td>
<td>#x04FF</td>
<td>Cyrillic</td>
<td>#x0530</td>
<td>#x058F</td>
<td>Armenian</td>
</tr>
<tr>
<td>#x0590</td>
<td>#x05FF</td>
<td>Hebrew</td>
<td>#x0600</td>
<td>#x06FF</td>
<td>Arabic</td>
</tr>
<tr>
<td>#x0700</td>
<td>#x074F</td>
<td>Syriac</td>
<td>#x0780</td>
<td>#x07BF</td>
<td>Thaana</td>
</tr>
<tr>
<td>#x0900</td>
<td>#x097F</td>
<td>Devanagari</td>
<td>#x0980</td>
<td>#x09FF</td>
<td>Bengali</td>
</tr>
<tr>
<td>#x0A00</td>
<td>#x0A7F</td>
<td>Gurmukhi</td>
<td>#x0A80</td>
<td>#x0AFF</td>
<td>Gujarati</td>
</tr>
<tr>
<td>#x0B00</td>
<td>#x0B7F</td>
<td>Oriya</td>
<td>#x0B80</td>
<td>#x0BFF</td>
<td>Tamil</td>
</tr>
<tr>
<td>#x0C00</td>
<td>#x0C7F</td>
<td>Telugu</td>
<td>#x0C80</td>
<td>#x0CFF</td>
<td>Kannada</td>
</tr>
<tr>
<td>#x0D00</td>
<td>#x0D7F</td>
<td>Malayalam</td>
<td>#x0D80</td>
<td>#x0DFF</td>
<td>Sinhala</td>
</tr>
<tr>
<td>#x0E00</td>
<td>#x0E7F</td>
<td>Thai</td>
<td>#x0E80</td>
<td>#x0EFF</td>
<td>Lao</td>
</tr>
<tr>
<td>Code</td>
<td>Code</td>
<td>Name</td>
<td>Code</td>
<td>Code</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>-------------------------------------------</td>
<td>---------</td>
<td>------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>#x000</td>
<td>#x0FF</td>
<td>Tibetan</td>
<td>#x1000</td>
<td>#x109F</td>
<td>Myanmar</td>
</tr>
<tr>
<td>#x10A0</td>
<td>#x10FF</td>
<td>Georgian</td>
<td>#x1100</td>
<td>#x11FF</td>
<td>HangulJamo</td>
</tr>
<tr>
<td>#x1200</td>
<td>#x137F</td>
<td>Ethiopic</td>
<td>#x13A0</td>
<td>#x13FF</td>
<td>Cherokee</td>
</tr>
<tr>
<td>#x1400</td>
<td>#x167F</td>
<td>UnifiedCanadianAboriginalSyllabics</td>
<td>#x1680</td>
<td>#x169F</td>
<td>Ogham</td>
</tr>
<tr>
<td>#x16A0</td>
<td>#x16FF</td>
<td>Runic</td>
<td>#x1780</td>
<td>#x17FF</td>
<td>Khmer</td>
</tr>
<tr>
<td>#x1800</td>
<td>#x18AF</td>
<td>Mongolian</td>
<td>#x1E00</td>
<td>#x1EFF</td>
<td>LatinExtendedAdditional</td>
</tr>
<tr>
<td>#x1F00</td>
<td>#x1FF</td>
<td>GreekExtended</td>
<td>#x2000</td>
<td>#x206F</td>
<td>GeneralPunctuation</td>
</tr>
<tr>
<td>#x2070</td>
<td>#x209F</td>
<td>SuperscriptsandSubscripts</td>
<td>#x20A0</td>
<td>#x20CF</td>
<td>CurrencySymbols</td>
</tr>
<tr>
<td>#x20D0</td>
<td>#x20FF</td>
<td>CombiningMarksforSymbols</td>
<td>#x2100</td>
<td>#x214F</td>
<td>LetterlikeSymbols</td>
</tr>
<tr>
<td>#x2150</td>
<td>#x218F</td>
<td>NumberForms</td>
<td>#x2190</td>
<td>#x21FF</td>
<td>Arrows</td>
</tr>
<tr>
<td>#x2200</td>
<td>#x22FF</td>
<td>MathematicalOperators</td>
<td>#x2300</td>
<td>#x23FF</td>
<td>MiscellaneousTechnical</td>
</tr>
<tr>
<td>#x2400</td>
<td>#x243F</td>
<td>ControlPictures</td>
<td>#x2440</td>
<td>#x245F</td>
<td>OpticalCharacterRecognition</td>
</tr>
<tr>
<td>#x2460</td>
<td>#x24FF</td>
<td>EnclosedAlphanumerics</td>
<td>#x2500</td>
<td>#x257F</td>
<td>BoxDrawing</td>
</tr>
<tr>
<td>#x2580</td>
<td>#x259F</td>
<td>BlockElements</td>
<td>#x25A0</td>
<td>#x25FF</td>
<td>GeometricShapes</td>
</tr>
<tr>
<td>#x2600</td>
<td>#x26FF</td>
<td>MiscellaneousSymbols</td>
<td>#x2700</td>
<td>#x27BF</td>
<td>Dingbats</td>
</tr>
<tr>
<td>#x2800</td>
<td>#x28FF</td>
<td>BraillePatterns</td>
<td>#x2E80</td>
<td>#x2EFF</td>
<td>CJKRadicalsSupplement</td>
</tr>
<tr>
<td>#x2F00</td>
<td>#x2FDF</td>
<td>KangxiRadicals</td>
<td>#x2FF0</td>
<td>#x2FFF</td>
<td>IdeographicDescriptionCharacters</td>
</tr>
</tbody>
</table>

F.1 Character Classes
<table>
<thead>
<tr>
<th>Code</th>
<th>Code</th>
<th>Description</th>
<th>Code</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>x303F</td>
<td>CJKSymbolsAndPunctuation</td>
<td>x3040</td>
<td>x309F</td>
<td>Hiragana</td>
</tr>
<tr>
<td>x30A0</td>
<td>x30FF</td>
<td>Katakana</td>
<td>x3100</td>
<td>x312F</td>
<td>Bopomofo</td>
</tr>
<tr>
<td>x3130</td>
<td>x318F</td>
<td>HangulCompatibilityJamo</td>
<td>x3190</td>
<td>x319F</td>
<td>Kanbun</td>
</tr>
<tr>
<td>x31A0</td>
<td>x31BF</td>
<td>BopomofoExtended</td>
<td>x3200</td>
<td>x32FF</td>
<td>EnclosedCJKLettersAndMonths</td>
</tr>
<tr>
<td>x3300</td>
<td>x33FF</td>
<td>CJKCompatibility</td>
<td>x3400</td>
<td>x4DB5</td>
<td>CJKUnifiedIdeographsExtensionA</td>
</tr>
<tr>
<td>x4E00</td>
<td>x9FFF</td>
<td>CJKUnifiedIdeographs</td>
<td>xA000</td>
<td>xA48F</td>
<td>YiSyllables</td>
</tr>
<tr>
<td>xA490</td>
<td>xA4CF</td>
<td>YiRadicals</td>
<td>xAC00</td>
<td>xD7A3</td>
<td>HangulSyllables</td>
</tr>
<tr>
<td>xD800</td>
<td>xDB7F</td>
<td>HighSurrogates</td>
<td>xDB80</td>
<td>xDBFF</td>
<td>HighPrivateUseSurrogates</td>
</tr>
<tr>
<td>xDC00</td>
<td>xDFFF</td>
<td>LowSurrogates</td>
<td>xE000</td>
<td>xF8FF</td>
<td>PrivateUse</td>
</tr>
<tr>
<td>xF900</td>
<td>xFAFF</td>
<td>CJKCompatibilityIdeographs</td>
<td>xFB00</td>
<td>xFB4F</td>
<td>AlphabeticPresentationForms</td>
</tr>
<tr>
<td>xFB50</td>
<td>xFDDF</td>
<td>ArabicPresentationForms−A</td>
<td>xFE20</td>
<td>xFE2F</td>
<td>CombiningHalfMarks</td>
</tr>
<tr>
<td>xFE30</td>
<td>xFE4F</td>
<td>CJKCompatibilityForms</td>
<td>xFE50</td>
<td>xFE6F</td>
<td>SmallFormVariants</td>
</tr>
<tr>
<td>xFE70</td>
<td>xFEFE</td>
<td>ArabicPresentationForms−B</td>
<td>xFEFF</td>
<td>xFEFF</td>
<td>Specials</td>
</tr>
<tr>
<td>xFF00</td>
<td>xFFEF</td>
<td>HalfwidthAndFullwidthForms</td>
<td>xFF00</td>
<td>xFFFD</td>
<td>Specials</td>
</tr>
<tr>
<td>x10300</td>
<td>x1032F</td>
<td>OldItalic</td>
<td>x10330</td>
<td>x1034F</td>
<td>Gothic</td>
</tr>
<tr>
<td>x10400</td>
<td>x1044F</td>
<td>Deseret</td>
<td>x1D000</td>
<td>x1D0FF</td>
<td>ByzantineMusicalSymbols</td>
</tr>
<tr>
<td>x1D100</td>
<td>x1D1FF</td>
<td>MusicalSymbols</td>
<td>x1D400</td>
<td>x1D7FF</td>
<td>MathematicalAlphanumericSymbols</td>
</tr>
</tbody>
</table>

F.1 Character Classes 328
NOTE: [Unicode Database] is subject to future revision. For example, the grouping of code points into blocks might be updated. All minimally conforming processors must support the blocks defined in the version of [Unicode Database] that is current at the time this specification became a W3C Recommendation. However, implementors are encouraged to support the blocks defined in any future version of the Unicode Standard.

For example, the block escape for identifying the ASCII characters is \p{IsBasicLatin}.

[Definition:] A multi-character escape provides a simple way to identify a commonly used set of characters:

```
Multi–Character Escape

[37] MultiCharEsc ::= ',' | ('\' [sSiIcCdDwW])
```

NOTE: The regular expression language defined here does not attempt to provide a general solution to "regular expressions" over UCS character sequences. In particular, it does not easily provide for matching sequences of base characters and combining marks. The language is targeted at support of "Level 1" features as defined in [Unicode Regular Expression Guidelines]. It is hoped that future versions of this specification will provide support for "Level 2" features.

G Glossary (non-normative)

The listing below is for the benefit of readers of a printed version of this document: it collects together all the definitions which appear in the document above.

atomic

Atomic datatypes are those having values which are regarded by this specification as being indivisible.

base type

Every datatype that is derived by restriction is defined in terms of an existing datatype, referred to as its base type. base types can be either primitive or derived.

bounded

A datatype is bounded if its value space has either an inclusive upper bound or an exclusive upper bound, and either an inclusive lower bound and an exclusive lower bound.

built-in

Built-in datatypes are those which are defined in this specification, and can be either primitive or derived.

canonical lexical representation
A **canonical lexical representation** is a set of literals from among the valid set of literals for a datatype such that there is a one-to-one mapping between literals in the **canonical lexical representation** and values in the **value space**.

**cardinality**

Every **value space** has associated with it the concept of **cardinality**. Some **value space**s are finite, some are countably infinite while still others could conceivably be uncountably infinite (although no **value space** defined by this specification is uncountable infinite). A datatype is said to have the cardinality of its **value space**.

**conformance to the XML Representation of Schemas**

Processors which accept schemas in the form of XML documents as described in XML Representation of Simple Type Definition Schema Components (§4.1.2) (and other relevant portions of Datatype components (§4)) are additionally said to provide **conformance to the XML Representation of Schemas**, and **must**, when processing schema documents, completely and correctly implement all **Schema Representation Constraint**-s in this specification, and **must** adhere exactly to the specifications in XML Representation of Simple Type Definition Schema Components (§4.1.2) (and other relevant portions of Datatype components (§4)) for mapping the contents of such documents to **schema components** for use in validation.

**constraining facet**

A **constraining facet** is an optional property that can be applied to a datatype to constrain its **value space**.

**Constraint on Schemas**

**Constraint on Schemas**

**datatype**

In this specification, a **datatype** is a 3-tuple, consisting of a) a set of distinct values, called its **value space**, b) a set of lexical representations, called its **lexical space**, and c) a set of **facet**s that characterize properties of the **value space**, individual values or lexical items.

**derived**

**Derived** datatypes are those that are defined in terms of other datatypes.

**error**

**error**

**exclusive lower bound**

A value \( l \) in an **ordered** **value space** \( L \) is said to be an **exclusive lower bound** of a **value space** \( V \) (where \( V \) is a subset of \( L \)) if for all \( v \) in \( V \), \( l < v \).

**exclusive upper bound**

A value \( u \) in an **ordered** **value space** \( U \) is said to be an **exclusive upper bound** of a **value space** \( V \) (where \( V \) is a subset of \( U \)) if for all \( v \) in \( V \), \( u > v \).

**facet**

A **facet** is a single defining aspect of a **value space**. Generally speaking, each facet characterizes a **value space** along independent axes or dimensions.

**for compatibility**

**for compatibility**

**fundamental facet**

A **fundamental facet** is an abstract property which serves to semantically characterize the values in a **value space**.

**inclusive lower bound**

A value \( l \) in an **ordered** **value space** \( L \) is said to be an **inclusive lower bound** of a **value space** \( V \) (where \( V \) is a subset of \( L \)) if for all \( v \) in \( V \), \( l \leq v \).

**inclusive upper bound**

A value \( u \) in an **ordered** **value space** \( U \) is said to be an **inclusive upper bound** of a **value space** \( V \) (where \( V \) is a subset of \( U \)) if for all \( v \) in \( V \), \( u \geq v \).

**itemType**
The _atomic_ datatype that participates in the definition of a _list_ datatype is known as the **itemType** of that _list_ datatype.

**lexical space**

A lexical space is the set of valid literals for a datatype.

**list**

List datatypes are those having values each of which consists of a finite-length (possibly empty) sequence of values of an _atomic_ datatype.

**match**

match

**may**

may

**memberTypes**

The datatypes that participate in the definition of a _union_ datatype are known as the **memberTypes** of that _union_ datatype.

**minimally conforming**

Minimally conforming processors **must** completely and correctly implement the **Constraint on Schemas** and **Validation Rule**.

**must**

must

**non-numeric**

A datatype whose values are not _numeric_ is said to be **non-numeric**.

**numeric**

A datatype is said to be **numeric** if its values are conceptually quantities (in some mathematical number system).

**ordered**

A _value space_, and hence a datatype, is said to be **ordered** if there exists an _order-relation_ defined for that _value space_.

**order-relation**

An order relation on a _value space_ is a mathematical relation that imposes a _total order_ or a _partial order_ on the members of the _value space_.

**partial order**

A partial order is an _order-relation_ that is irreflexive, asymmetric and transitive.

**primitive**

Primitive datatypes are those that are not defined in terms of other datatypes; they exist ab initio.

**regular expression**

A regular expression is composed from zero or more _branch_ es, separated by | characters.

**restriction**

A datatype is said to be _derived_ by restriction from another datatype when values for zero or more _constraining facet_ s are specified that serve to constrain its _value space_ and/or its _lexical space_ to a subset of those of its _base type_.

**Schema Representation Constraint**

Schema Representation Constraint

**total order**

A total order is an _partial order_ such that for no a and b is it the case that a < > b.

**union**

Union datatypes are those whose _value space_ s and _lexical space_ s are the union of the _value space_ s and _lexical space_ s of one or more other datatypes.

**user-derived**

User-derived datatypes are those _derived_ datatypes that are defined by individual schema designers.

**Validation Rule**

Validation Rule
value space
A value space is the set of values for a given datatype. Each value in the value space of a datatype is denoted by one or more literals in its lexical space.

H References

H.1 Normative

Clinger, WD (1990)
ftp://ftp.ccs.neu.edu/pub/people/will/howtoread.ps

IEEE 754–1985
IEEE. IEEE Standard for Binary Floating–Point Arithmetic. See

Namespaces in XML
World Wide Web Consortium. Namespaces in XML. Available at:
http://www.w3.org/TR/1999/REC-xml-names-19990114/

RFC 1766
H. Alvestrand, ed. RFC 1766: Tags for the Identification of Languages 1995. Available at:
http://www.ietf.org/rfc/rfc1766.txt

RFC 2045

RFC 2396

RFC 2732
RFC 2732: Format for Literal IPv6 Addresses in URL’s. 1999. Available at:
http://www.ietf.org/rfc/rfc2732.txt

Unicode Database
The Unicode Consortium. The Unicode Character Database. Available at:
http://www.unicode.org/Public/3.1-Update/UnicodeCharacterDatabase-3.1.0.html

XML 1.0 (Second Edition)
http://www.w3.org/TR/2000/WD-xml-2e-20000814

XML Linking Language
World Wide Web Consortium. XML Linking Language (XLink). Available at:
http://www.w3.org/TR/2000/PR-xlink-20001220/

XML Schema Part 1: Structures
XML Schema Part 1: Structures. Available at:
http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

XML Schema Requirements
World Wide Web Consortium. XML Schema Requirements. Available at:
http://www.w3.org/TR/1999/NOTE-xml-schema-req-19990215

H.2 Non−normative

Character Model
XML Schema Part 0: Primer


Gay, DM (1990)

HTML 4.01

IETF INTERNET−DRAFT: IRIs

International Earth Rotation Service (IERS)
International Earth Rotation Service (IERS). See http://maia.usno.navy.mil

ISO 11404

ISO 8601

ISO 8601 Draft Revision

Perl

RDF Schema

Ruby

SQL

U.S. Naval Observatory Time Service Department
Information about Leap Seconds Available at: http://tycho.usno.navy.mil/leapsec.990505.html

Unicode Regular Expression Guidelines

XML Schema Language: Part 2 Primer

XSL
I Acknowledgements (non–normative)

The following have contributed material to this draft:

- Asir S. Vedamuthu, webMethods, Inc
- Mark Davis, IBM

Co–editor Ashok Malhotra's work on this specification from March 1999 until February 2001 was supported by IBM.

The editors acknowledge the members of the XML Schema Working Group, the members of other W3C Working Groups, and industry experts in other forums who have contributed directly or indirectly to the process or content of creating this document. The Working Group is particularly grateful to Lotus Development Corp. and IBM for providing teleconferencing facilities.

The current members of the XML Schema Working Group are:

Jim Barnette, Defense Information Systems Agency (DISA); Paul V. Biron, Health Level Seven; Don Box, DevelopMentor; Allen Brown, Microsoft; Lee Buck, TIBCO Extensibility; Charles E. Campbell, Informix; Wayne Carr, Intel; Peter Chen, Bootstrap Alliance and LSU; David Cleary, Progress Software; Dan Connolly, W3C (staff contact); Ugo Corda, Xerox; Roger L. Costello, MITRE; Haavard Danielson, Progress Software; Josef Dietl, Mozquito Technologies; David Ezell, Hewlett–Packard Company; Alexander Falk, Altova GmbH; David Fallside, IBM; Dan Fox, Defense Logistics Information Service (DLIS); Matthew Fuchs, Commerce One; Andrew Goodchild, Distributed Systems Technology Centre (DSTC Pty Ltd); Paul Grosso, Arbortext, Inc; Martin Gudgin, DevelopMentor; Dave Hollander, Contivo, Inc (co–chair); Mary Holstege, Invited Expert; Jane Hunter, Distributed Systems Technology Centre (DSTC Pty Ltd); Rick Jelliffe, Academia Sinica; Simon Johnston, Rational Software; Bob Lojek, Mozquito Technologies; Ashok Malhotra, Microsoft; Lisa Martin, IBM; Noah Mendelsohn, Lotus Development Corporation; Adrian Michel, Commerce One; Alex Milowski, Invited Expert; Rob Ellman, Calico Commerce; George Feinberg, Object Design; Charles Frankston, Microsoft; Ernesto Guerrieri, Inso; Michael Hyman, Microsoft; Renato Iannella, Distributed Systems Technology Centre (DSTC Pty Ltd); Dianne Kennedy, Graphic Communications Association; Jonathan Robie, Software AG; Eric Sedlar, Oracle Corp.; C. M. Sperberg–McQueen, W3C (co–chair); Bob Streich, Calico Commerce; William K. Stumbo, Xerox; Henry S. Thompson, University of Edinburgh; Mark Tucker, Health Level Seven; Asir S. Vedamuthu, webMethods, Inc; Priscilla Walmsley, XMLSolutions; Norm Walsh, Sun Microsystems; Aki Yoshida, SAP AG; Kongyi Zhou, Oracle Corp.

The XML Schema Working Group has benefited in its work from the participation and contributions of a number of people not currently members of the Working Group, including in particular those named below. Affiliations given are those current at the time of their work with the WG.

Paula Angerstein, Vignette Corporation; David Beech, Oracle Corp.; Gabe Beged–Dov, Rogue Wave Software; Greg Bumgardner, Rogue Wave Software; Dean Burson, Lotus Development Corporation; Mike Cokus, MITRE; Andrew Eisenberg, Progress Software; Rob Ellman, Calico Commerce; George Feinberg, Object Design; Charles Frankston, Microsoft; Ernesto Guerrieri, Inso; Michael Hyman, Microsoft; Renato Iannella, Distributed Systems Technology Centre (DSTC Pty Ltd); Dianne Kennedy, Graphic Communications Association; Janet Koenig, Sun Microsystems; Setrag Khoshaian, Technology Deployment International (TDI); Ara Kullukian, Technology Deployment International (TDI); Andrew Layman, Microsoft; Dmitry Lenkov, Hewlett–Packard Company; John McCarthy, Lawrence Berkeley National Laboratory; Murata Makoto, Xerox; Eve Maler, Sun Microsystems; Murray Maloney, Muzmo Communication, acting for Commerce One; Chris Olds, Wall Data; Frank Olken, Lawrence Berkeley National Laboratory; Shriram Revankar, Xerox; Mark Reinhold, Sun Microsystems; John C. Schneider,
XML Schema Requirements
W3C Note 15 February 1999

This version: http://www.w3.org/TR/1999/NOTE-xml-schema-req-19990215
Latest version: http://www.w3.org/TR/NOTE-xml-schema-req
Editors:
Ashok Malhotra (petsa@us.ibm.com) for IBM
Murray Maloney (murray@muzmo.com) for Veo Systems Inc.

Status of this document

This is a W3C Note published on 15 February 1999 as a deliverable of the XML Schema Working Group, which is part of the W3C XML Activity. It lists a base set of agreed requirements for an XML schema language.

This document represents a compromise that leaves many design questions open, creating opportunity for decision-making in the design phase. As the XML Schema work continues, the concrete implications of these requirements for the design will be worked out and documented. This document is a living document; it will be reviewed regularly by the Working Group, and may be revised to reflect changes in the Working Group's understanding. The Working Group does not anticipate substantial changes, but may decide to refine existing requirements or add new ones.

Comments about this document should be addressed to the XML Schema Requirements Comments list at www-xml-schema-comments@w3.org. Comments accumulated by 1 March 1999 will be reviewed in March 1999.

A list of current W3C technical reports and publications, including working drafts and notes, can be found at http://www.w3.org/TR.

Abstract

This document specifies the purpose, basic usage scenarios, design principles, and base requirements for an XML schema language.

Table of Contents

1. Overview
2. Purpose
3. Scenarios
4. Principles
5. Requirements
   ♦ Structural schemas
   ♦ Datatypes
   ♦ Conformance
1. Overview

The XML 1.0 specification defines the concepts of well-formedness and validity; it is very simple to check a document for well-formedness, while validation requires more work but allows the user to define more powerful constraints on document structure. XML validity requires that a document follow the constraints expressed in its document type definition, which provides the rough equivalent of a context-free grammar for a document type.

For some uses, applications may need definitions of markup constructs more informative, or constraints on document structure tighter than, looser than, or simply different from those which can be expressed using document type definitions as defined in XML 1.0. There is also a widespread desire to allow markup constructs and constraints to be specified in an XML-based syntax, in order to allow tools for XML documents to be used on the specifications.

By charter, the XML Schema Working Group is assigned to address the following issues:

**structural schemas**
- a mechanism somewhat analogous to DTDs for constraining document structure (order, occurrence of elements, attributes). Specific goals beyond DTD functionality are
  - integration with namespaces
  - definition of incomplete constraints on the content of an element type
  - integration of structural schemas with primitive data types
  - inheritance: Existing mechanisms use content models to specify part-of relations. But they only specify kind-of relations implicitly or informally. Making kind-of relations explicit would make both understanding and maintenance easier

**primitive data typing**
- integers, dates, and the like, based on experience with SQL, Java primitives, etc.; byte sequences ("binary data") also need to be considered

**conformance**
- The relation of schemata to XML document instances, and obligations on schema-aware processors, must be defined. The Working Group will define a process for checking to see that the constraints expressed in a schema are obeyed in a document (schema-validation); the relationship between schema-validity and validity as defined in XML 1.0 will be defined.

The XML Schema work is interdependent with several other areas of W3C activity. These are listed below under Design Principles.

2. Purpose

The purpose of the XML schema language is to provide an inventory of XML markup constructs with which to write schemas.

The purpose of a schema is to define and describe a class of XML documents by using these constructs to constrain and document the meaning, usage and relationships of their constituent parts: datatypes, elements and their content, attributes and their values, entities and their contents and notations. Schema constructs may also provide for the specification of implicit information such as default values. Schemas document their own meaning, usage, and function.

Thus, the XML schema language can be used to define, describe and catalogue XML vocabularies for classes of XML documents.
Any application of XML can use the Schema formalism to express syntactic, structural and value constraints applicable to its document instances. The Schema formalism will allow a useful level of constraint checking to be described and validated for a wide spectrum of XML applications. For applications which require other, arbitrary or complicated constraints, the application must perform its own additional validations.

3. Usage Scenarios

The following usage scenarios describe XML applications that should benefit from XML schemas. They represent a wide range of activities and needs that are representative of the problem space to be addressed. They are intended to be used during the development of XML schemas as design cases that should be reviewed when critical decisions are made. These usage scenarios should also prove useful in helping non-members of the XML Schema Working Group understand the intent and goals of the project.

1. Publishing and syndication

Distribution of information through publishing and syndication services. Involves collections of XML documents with complex relations among them. Structural schemas describe the properties of headlines, news stories, thumbnail images, cross-references, etc. Document views under control of different versions of a schema.

2. Electronic commerce transaction processing.

Libraries of schemas define business transactions within markets and between parties. A schema-aware processor is used to validate a business document, and to provide access to its information set.

3. Supervisory control and data acquisition.

The management and use of network devices involves the exchange of data and control messages. Schemas can be used by a server to ensure outgoing message validity, or by the client to allow it to determine what part of a message it understands. In multi-vendor environment, discriminates data governed by different schemas (industry-standard, vendor-specific) and know when it is safe to ignore information not understood and when an error should be raised instead; provide transparency control. Applications include media devices, security systems, plant automation, process control.

4. Traditional document authoring/editing governed by schema constraints.

One important class of application uses a schema definition to guide an author in the development of documents. A simple example might be a memo, whereas a more sophisticated example is the technical service manuals for a wide-body intercontinental aircraft. The application can ensure that the author always knows whether to enter a date or a part-number, and might even ensure that the data entered is valid.

5. Use schema to help query formulation and optimization.

A query interface inspect XML schemas to guide a user in the formulation of queries. Any given database can emit a schema of itself to inform other systems what counts as legitimate and useful queries.

6. Open and uniform transfer of data between applications, including databases
XML has become a widely used format for encoding data (including metadata and control data) for exchange between loosely coupled applications. Such exchange is currently hampered by the difficulty of fully describing the exchange data model in terms of XML DTDs; exchange data model versioning issues further complicate such interactions. When the exchange data model is represented by the more expressive XML Schema definitions, the task of mapping the exchange data model to and from application internal data models will be simplified.

7. Metadata Interchange

There is growing interest in the interchange of metadata (especially for databases) and in the use of metadata registries to facilitate interoperability of database design, DBMS, query, user interface, data warehousing, and report generation tools. Examples include ISO 11179 and ANSI X3.285 data registry standards, and OMG’s proposed XMI standard.

4. Design Principles

In the design of any language, trade-offs in the solution space are necessary. The following design principles should guide the working group in making these trade-offs. Design principles are desirable, but not fully measurable, characteristics.

The XML schema language shall be:

1. more expressive than XML DTDs;
2. expressed in XML;
3. self-describing;
4. usable by a wide variety of applications that employ XML;
5. straightforwardly usable on the Internet;
6. optimized for interoperability;
7. simple enough to implement with modest design and runtime resources;
8. coordinated with relevant W3C specs (XML Information Set, Links, Namespaces, Pointers, Style and Syntax, as well as DOM, HTML, and RDF Schema).

The XML schema language specification shall:

1. be prepared quickly;
2. be precise, concise, human-readable, and illustrated with examples.

5. Requirements

Structural requirements

The XML schema language must define:

1. mechanisms for constraining document structure (namespaces, elements, attributes) and content (datatypes, entities, notations);
2. mechanisms to enable inheritance for element, attribute, and datatype definitions;
3. mechanism for URI reference to standard semantic understanding of a construct;
4. mechanism for embedded documentation;
5. mechanism for application-specific constraints and descriptions;
6. mechanisms for addressing the evolution of schemata;
7. mechanisms to enable integration of structural schemas with primitive data types.

**Datatype requirements**

The XML schema language must:

1. provide for primitive data typing, including byte, date, integer, sequence, SQL & Java primitive data types, etc.;
2. define a type system that is adequate for import/export from database systems (e.g., relational, object, OLAP);
3. distinguish requirements relating to lexical data representation vs. those governing an underlying information set;
4. allow creation of user-defined datatypes, such as datatypes that are derived from existing datatypes and which may constrain certain of its properties (e.g., range, precision, length, mask).

**Conformance**

The XML schema language must:

1. describe the responsibilities of conforming processors;
2. define the relationship between schemas and XML documents;
3. define the relationship between schema validity and XML validity;
4. define the relationship between schemas and XML DTDs, and their information sets;
5. define the relationship among schemas, namespaces, and validity;
6. define a useful XML schema for XML schemas;